Entwicklung effizienter Ranking Algorithmen für den Oracle Dienst

Developing Efficient Ranking Algorithms for the Oracle Service

Author: Benjamin Frank
Supervisor: Prof. Dr Thomas Fuhrmann
Advisor: Benedikt Elser, Obi Akonjang
Submission Date: August 15, 2009
I assure the single handed composition of this master’s thesis only supported by declared resources.

Date: 14.08.2009
Signature:
# Contents

1 Motivation 2

2 Introduction to the Oracle 3
   2.1 The Oracle Concept 3
   2.2 Structured and Unstructured Peer-2-Peer Systems 4
   2.3 The Oracle Protocol 5
   2.4 Related Works 6

3 Previous Work (Gnutella and Oracle) 8

4 The BitTorrent Protocol 11
   4.1 Encoding 11
   4.2 The Metafile (.torrent) File 11
   4.3 The Tracker 12
   4.4 The Peerwire Protocol 13
      4.4.1 data-types and message flow 14
      4.4.2 messages 14
   4.5 Choke and Unchoke Algorithms 15
   4.6 Piece Downloading Strategy 16
   4.7 Differences to Gnutella 17
   4.8 Use of the Oracle 17

5 Simulation Environment and other used Tools 18
   5.1 SSFNet Simulation Environment 18
   5.2 Domain Modeling Language 18
   5.3 BitTorrent-Tracker Implementation 19
   5.4 BitTorrent-Client Implementation 20
   5.5 The Oracle Implementation 24
   5.6 Used Tools 25
   5.7 Script Framework 25

6 System Design and Metrics 29
   6.1 The Network Model 29
   6.2 The Ranking Function 31
   6.3 Influential Parameters 32
   6.4 Evaluation Metrics 33

7 Results and Evaluation 35
   7.1 Case 1: Congested peering links with seed in own network 38
   7.2 Case 2: Congested peering links with seed in Tier-3 network 41
   7.3 Case 3: Uncongested peering links 44
   7.4 Case 4: Overprovisioned seed 47
   7.5 Case 5: Multiple torrents with seeds in our network 51

8 Conclusion and Future Work 54
1 Motivation

Bandwidth demanding applications like Peer-to-Peer file sharing applications are often cited as the major reason for end users desire to upgrade their Internet access to increasingly faster broadband and as such are the mainspring for increasing the ISP's revenue [2]. Recent studies have shown that today's Internet traffic is composed of 50-65% Peer-to-Peer data transfers, 20-30% HTTP webtraffic and roughly 10% streaming media [28]. A more conservative estimate still sees up to 25% of Peer-to-Peer traffic [44] with a much higher proportion of web traffic.

While Peer-to-Peer and streaming media are lucrative for the ISPs because it drives people to upgrade their internet access on a regular basis due to the ever increasing bandwidth demands, it has also introduced severe problems for the network operators as it poses a significant traffic engineering challenge [33]. This is because most of the Peer-to-Peer systems used today rely on an application layer routing based on the overlay topology which is largely agnostic of the routig in the underlay [20] and thus even contradicts the efforts done to optimize the performance in the underlaying network. It has been shown that Peer-to-Peer traffic traverses network boundaries multiple times before arriving at its final destination [20, 32]. This might lead to a situation where a peer in such a file sharing system tries to download content from a peer that is literally on the other end of the world although the content would also be available from a peer in the same network. As a result the generated traffic traverses multiple networks in between, stressing the already present peering bottlenecks [4] even further, traffic that could be saved if the peers would know of each other. This local availability of content is due to language and geographical areas of interest as found by numerous studies [7, 32].

To ease this situation Vinay Aggarwal and Anja Feldmann proposed the Oracle service [2] which acts as an Interface between the overlay and the underlay network to allow an optimization of the overelay routing decisions. To achieve this a peer has to send a list of candidate peers it wants to interconnect with to the Oracle which then reorders this list to its best knowledge in respect to locality of the client and the peers. The Oracle server is operated by the network provider offering the service and thus has a broad knowledge of network properties that otherwise would be confidential to the end-user. Because the Oracle server uses this information only to reorder the list of possible candidate peers for its clients no confidential information about the network is revealed to the clients at any time. Earlier experiments done with Gnutella [2] have already shown the potential gains the Oracle can enable in such a Peer-to-Peer system by optimizing the peer connections in the overlay network with respect to their location and distance in the underlaying network. But the landscape of Internet traffic has changed quite a lot since then and BitTorrent has become the dominant Peer-to-Peer file sharing protocol with an estimated proportion of up to 30% of the overall Internet traffic [28] today.

Subsequently this thesis expands the scope of the Oracle to include further experiments with the popular BitTorrent Peer-to-Peer file sharing protocol and expands the scope of parameters to see the effects they could have on the observed results. This thesis thus will provide the first practical steps for tailoring efficient Oracle ranking functions suited for the individual needs of network providers.
2 Introduction to the Oracle

In this chapter the Oracle service developed by Vinay Aggarwal and Anja Feldmann in [2] will be introduced and its expected results explained. The purpose of the Oracle service is twofold: for the end-user, that is the people seeking the Oracles assistance, the service will, after recieving a list of possible candidate peers, provide them with a ranked list of those peers reflecting the relative connectivity of the client to those peers, resulting in better performance for the end-user. For an ISP the Oracle yields the possibility to influence the neighbourhood selection strategy of a peer-to-peer system or any other system that has multiple possible candidates for the offered service and therefore regaining the ability of traffic engineering in underlay agnostic overlay networks. The Oracle service shares a lot of similarities with the classic DNS servers. It provides a basic information base which improves the perceived quality of a user. While the DNS system converts human readable and therefore better understandable names into IP addresses, the Oracle sorts a given IP list for a client such that the individual connections to those IPs will yield a better performance for the client. None of the services is really necessary, but both add quite a huge qualitative aspect to the users perceived experience. To achieve this they both need to be fast and reliable as well as scalable. Thus many concepts and properties of the DNS protocol were reused in development of the Oracle protocol.

2.1 The Oracle Concept

Figure 1 shows the basic steps necessary for using the Oracle service. Whenever a client needs to decide which one of a list of peers it wants to connect with this list is sent to the Oracle server (1). The Oracle server then ranks every peer in that list with respect to its connectivity to the client asking for the Oracle’s assistance and re-orders the list to match this ranking (2). The list is then sent back to the client (3) which may now choose to follow this recommendation to take full benefit of the Oracles knowledge of the underlay network structure (4). While herein the term connectivity is not further explained or even exactly defined, the Oracle can use a wide variety of factors to determine the connectivity of a peer. This might for instance be the upload capacity of the remote peer, the inter-AS distance between those two peers or even the expected round trip time between them. The exact function used to determine the connectivity from all available factors is the subject of this work and will be explained in detail later in this thesis.
With the information provided by the Oracle a peer can expect to see its performance increase while an ISP offering the Oracle service has the possibility to regain traffic engineering abilities and keep traffic as local as possible by promoting peers in its own network. Again the term performance is not extensively defined and may have different meaning for different applications. For a pure downloading scenario this would certainly mean higher download speeds but for a streaming application it might be lower round trip times combined with a lower jitter for a connection.

2.2 Structured and Unstructured Peer-2-Peer Systems

In most of the overlay networks used today neighbors are chosen randomly (unstructured overlays, like Gnutella), with help of a central peer management instance (hybrid overlays, like BitTorrent) or such that a certain pattern with connection between peers are achieved (structured overlays, like Chord). In the unstructured overlay networks, peer connections are established on an arbitrary basis, each peer can connect to whichever peer he likes. In those unstructured networks search queries pose the biggest problem as they have to be flooded into the network. This does not ensure that available content will be found as the query might not reach all peers due to protection measures against drowning the overlay network in flooded search queries. While popular content will be present in many peers and thus will be found quickly, less likened content with few peers offering the data will be harder to find in unstructured networks as a query might not reach those peers.

In hybrid overlay networks the connection between peers are also established on an arbitrary basis but a central management instance is queried for a list of possible peers that a client can connect to. Thus the bootstrapping of a peer in such an overlay network is much easier as a peer can ask for new peers but has the drawback of the additional management instance being available at all time. In BitTorrent that central management instance is called a tracker and answers requests for peers on a per file basis so that different files will create distinct overlay networks.

In structured networks a globally consistent algorithm is utilized to ensure an efficient routing of queries to peers having the requested content, no matter how popular or unpopular it might be. To achieve such a property most overlay networks employ a “Distributed Hash Table” (DHT) [18]. A DHT is a decentralized hashtable offering a similar lookup service like a traditionally hashtable: the hashtable stores (key, value)-pairs where the values can be retrieved efficiently for a given key. In a DHT each peer is responsible for a certain subset of keys, most often a certain range of keys from the complete keyspace\(^1\). If a peer is searching for content, the query is routed along the overlay to the node responsible for the respective key of that search. The routing within the structured overlay network is dependant on the actual implementation, examples would be: geographic routing like it is employed by “Content Addressable Network” (CAN) [43], a binary search like algorithm in a ring structure like it is done in Chord [53] or successive prefix matching in a hypercube like structure as done in Pastry [45].

While the structured approach helps to keep the clients interconnected due to a high clustering index, it does come with a certain cost. Most of the work done by the overlay is also done in the underlay and even contradicts the work done in the underlay, as it is agnostic to the underlays effort. This is best explained with one of the basic features that both networks provide: the routing. The overlay resides in a pure virtual space, a neighbour in that virtual space might be much further away in the real network than a far away node in the virtual space, which even might be a neighbour in the real network. This can happen because the position of a node in the virtual space of the overlay is chosen without any consideration of its location in the real network. Due to this the routing decision in the overlay might contradict the routing decision in the underlay. Now consider this scenario with multiple hops alternating between nodes outside the ISP’s network and inside, which would lead to a “bouncing” of packets in and out of the nodes ISP.

In the unstructured overlay network the help of the Oracle pretty simple, it allows a peer to choose connections to peers in the virtual space that are better in the sense of connectivity and locality to the client. Because the connections between peers are arbitrary this does not violate the nature of

\(^1\)Often a certain key range is replicated among many peers so that node churn, malicious users or data corruption can be dealt with.
unstructured overlay networks. There is a certain risk that emphasizing on local connections between peers to much might fragment the overlay network into multiple distinct overlay networks but recent studies have shown that the overlay stays well connected even if the Oracle focuses on local connections between peers (see chapter 3).

In structured overlay networks this becomes a bit more complicated, because the position of a node in the virtual space determines the connections it will open to other nodes. But in general a structured overlay network does allow a certain variation in those connections, which means a node can open a connection to a peer in a certain area of the virtual space instead of a specific node. This allows again a useful integration of the Oracle in the neighbour selection process, because a client can now pick the best node in that area of the keyspace.

2.3 The Oracle Protocol

A lot of similarities exist between the widely known Domain Name System (DNS) protocol and the here explained Oracle protocol. The DNS protocol served as a prototype for the Oracle protocol because of the fact that it is highly distributable and scalable as well as able to handle many simultaneous queries from many clients, which were the exact same properties needed for the Oracle’s approach explained in section 2.1. Just like DNS the Oracle uses UDP as its transport layer protocol because:

- UDP is faster and more efficient if no guaranteed delivery is needed
- UDP is stateless, which is beneficial for processing small requests from a large amount of clients
- UDP is favorable for time sensitive protocols, as there are no acknowledgements or retransmissions of lost datagrams

The Oracle protocol was designed with generic types and extensibility in mind, such that it will be easy to allow different types of data-items to be ranked by the Oracle. To achieve this all data-item of the same type sent in an Oracle message are preceded by a Type-Length-Value (TLV) field, specifying the type and size of the following data. This approaches enables the Oracle protocol to be extended beyond its current needs and requirements by simply defining new types of TLVs. In the first version of the protocol only IPv4/IPv6 addresses and prefixes are supported by the appropriate TLV header. For further details on the packet format or detailed information on TLV see [6]. An Oracle message can be seen in Figure 2 and generally consists of three sections: a fixed size header, a variable size extension header and the payload (in form of TLVs).

![Figure 2: Packet format scheme of an oracle message.](image)

The fixed size header contains the protocol version used in this request, a transaction ID to prevent unauthoritative answers, a TTL value giving the time the reply can be cached by the client/server, a service category field giving the generalized category of the application protocol in the payload\(^2\), and a

\(^2\)Examples may be Oracle TLVs, ISP-Driven Informed Path Selection (IDIPS) or other similar services.
field for different authentication types (in combination with an authentication extension header, see below). The header also contains various flags, especially noteworthy are the service code flags, which can set the requested peer ranking to favour bandwidth or delay. Current supported extension-headers are for authentication and IP source-addresses (in case a list of IP addresses is sent by a client/server on behalf of another client/server). The payload consists of one or more TLVs, each containing one or more data-items that the Oracle should rank for the client. A TLV specifies a type (Type) along with the size of the data in bytes (Size)\(^3\) and an optional attribute (AT) along with its length (AL), which could be an 2-letter country code or domain name, which might assist the Oracle in the ranking process. The next header field (NH) specifies if more TLVs follow after the actual one. The layout of a TLV can be seen in Figure 3.

![Figure 3: Layout of a Type-Length-Value (TLV) container](image)

### 2.4 Related Works

The idea of trying to make the overlay network aware of the underlay network and at least a few of its properties has already inspired a few people to implement services similar to the Oracle. Most of them use active measurement to determine those properties and as a side-effect add even more traffic and thus congestion to the network.

One of the early systems to measure performance between two nodes is Vivaldi [17]. It relies on measuring latencies between two nodes to calculate coordinates in an euclidean space for each node. Those coordinates are calculated by solving a spring relaxation problem, which requires to be successively refined by periodic updates that include re-measuring the latency between nodes. Although there is a possibility to piggyback the required data on the application protocol, this still induces overhead in active measurement.

Another approach uses available information from Content Distribution Networks (CDNs) to find optimized peer connections. One adopter of this technique is Ono [12], which reuses the network measurements by big public CDNs. To improve the end-user’s performance a CDN usually places the servers that deliver the content to its customers on the edge of an ISP’s network [3, 30]. Most CDNs place servers at ISP’s points of presence and use DNS-redirects or URL-rewrites to forward requests from the clients to topologically close servers. The fact that peers that get the same content server must be topologically close to each other is the foundation of this approach. The information about CDN servers thus is used to find peers that are close to each other, thereby improving their performance. Main problem of this approach is the reliance on data provided by 3rd parties, namely the CDNs, as well as the overhead introduced by the piggybacking of information. A CDN might not be present in every single ISP and thus skew the results for clients not having a topologically close CDN server, as they would be pointed to a server further away, while this approach would consider this client still close to the server. Another problem might be introduced by the measurements done by the CDN, as they might change their methodology without notice, thus skewing the results again. Also the possibility that a CDN might disappear, by bankruptcy or ceasing operation in certain parts of the world, has to be taken into consideration.

A quite different idea to improve the performance of peer-2-peer clients is the deployment of a local cache for peer-2-peer traffic by the ISP. pCache [27] is such an implementation of a p2p proxy cache, which intercepts request from p2p systems and tries to answer them from its cache. If no content is found in its cache, the request is forwarded normally, else the content is delivered directly from the

\(^3\)Currently only IPv4/v6 addresses and prefixes are supported, but 250 types remain undefined.
cache to the peers. This approach enables the ISP to avoid traffic on (possible expensive) peering links, but also introduces some problematic properties. The legal aspect might be the most important, as the cache might have whole copies of not licensed data. Another problematic aspect is the intercepting and answering of p2p requests by the cache. To answer a request, the cache has to impersonate the peer that was asked for the data, resulting in spoofing its IP address. This might lead to quite a few security and privacy implications. Finally this caching approach is undermined by recent development in the p2p community: communication encryption. With the use of encryption between two peers, the cache is not able to intercept requests and thus can not answer them, rendering it useless.

The last approach to be considered is the Provider Portal for P2P (P4P) [57] approach. The P4P framework consist of two components: the control-plane and the data-plane. The control plane provides detailed information about the network through iT rackers, creating a portal between the network provider and the users, which allows deviating the traffic control responsibility between them. To achieve this a network provider maintains one or more iT rackers and a client may obtain the IP address of its local provider through a DNS query, for which a new record type P4P is introduced. The information regarding the network provider an iT racker can provide are network status/topology, provider guidelines/policies and network capabilities, which the client can in turn use to determine optimal peer connectivity. To gather this information, the iT racker allows the internal routers of the network to provide fine-grained feedback to the data-plane. For extensibility all communication is done via XML messages. This approach has two major design decisions that may contradict the requirements for such a system. First, the network provider hands out detailed information about its network directly to its users, information that most network providers still classify as sensible and security critical information [38]. Second, the possible size of such information: to find optimal peering partners, all information about the network has to be available to the client and it has to be recent information, as the properties of a network are subject to frequent changes. This leads to the problem of maintaining the information provided by the P4P framework, as many p2p systems establish connections throughout their complete lifetime and not only in the beginning, which results in frequent updates pulled from the iT racker, as the different properties of a network are subject to quite frequent changes [4, 36]. Another aspect is that by just supplying this information, a client still has to implement its own optimization algorithm on the data, which is far from trivial, as different applications have different requirements and the implementation might even vary among different implementations of the very same protocol and thus even might contradict the optimization done by other clients.
3 Previous Work (Gnutella and Oracle)

The concept of the Oracle was first presented in 2007 by Anja Feldmann and Vinay Aggarwal from the Deutsche Telekom Laboratories at TU Berlin [2]. To explore how real peer-to-peer file sharing systems might benefit from a recommendation system the Oracle provides various simulations with SSFNet were conducted. The protocol chosen was Gnutella [25], an unstructured peer-to-peer file sharing protocol having roughly 2 million users by that time, that uses the created overlay network to allow its peers to search for downloaded content.

A peer joins the overlay network by sending a Ping message to one known participant of the overlay who answers with a Pong message containing additional peer addresses and shared resource information. Searching for content is done by flooding Query messages into the Gnutella network and are answers by QueryHit messages which travel back the reverse (overlay) path the request took. To limit flooding the network Gnutella uses TTL (time to live) values and message IDs (which also prevent loops). Thus the generated traffic in the overlay network of Gnutella by new peers joining and conducted searches is quite demanding. The actual download of content takes place outside of the created overlay network using the widely known HTTP protocol within a direct connection between two peers.

Gnutella has attracted many researchers [55, 24, 54] due to it being open source and the wide adoption by endusers. Due to scalability problems, newer versions of the protocol take advantage of an hierarchical design in which well connected nodes are promoted to ultrapeers. Normal peers connect to few ultrapeers while those connect to many peers, normal ones as well as other ultrapeers. Another improvement is the caching of Pong messages and the limitation in frequency of Ping and Query messages.

Gnutella was implemented for SSFNet and the neighbour selection procedure was changed to advantage of the proposed Oracle service [2]. Normally a Gnutella client randomly connects to a certain number of peers it has in its hostcache from earlier Gnutella sessions or retrieves a publically available list of peers in the Gnutella network. Instead of randomly connecting to other peers, the list of possible candidate peers is sent to the Oracle server which then picks a peer that is within the same AS. The client then establishes connections to multiple of this Oracle preferred peers and thus the Oracle influences the neighbourhood selection procedure of the Gnutella protocol. The same is done when query results for search requests are received to pick the closest peer that offers the desired content for downloading.

The network chosen for the simulations of the Gnutella network is described in detail in [2] and consists of 25 ASes divided into three tiers, 50 routers and 1000 nodes running the Gnutella protocol. More specifically it consists of 1 Tier-1 AS with 300 Gnutella peers, 8 Tier-2 ASes with 40 Gnutella peers each and 16 Tier-3 ASes each hosting 20 Gnutella peers. Within each AS a star topology connects all hosts to an intra-AS router, the host in the Tier-1 AS having a 1 GBit network link, the host in the Tier-2 ASes having 100 MBit links and the hosts in the Tier-3 networks have 10 Mbit links. The delay between the Tier-1 and Tier-2 networks is 2 ms while the delay between Tier-2 and Tier-3 networks is set to 10 ms.

The Gnutella clients are configured that a normal peer can have between 2 to 4 connections to ultrapeers while an ultrapeer will have at least 10 connections to other peers with a maximum of 45 peer connections. Each node shares between 0 and 100 uniformly distributed files. The time simulated was 5000 seconds as longer simulation times did not show significant changes.

The simulations were run in three different setups on five different topology instances, each having roughly the same number of search queries:

1. no Oracle service was used and the hostcache had a size of 1000 IPs
2. the Oracle service was used for neighbour selection and the hostcache had a size of 100 IPs
3. the Oracle service was used for neighbour selection and the hostcache had a size of 1000 IPs

SSFNet is a packet level network simulation environment and will be introduced in detail in chapter 5.1.
This is known as bootstrapping, to join at least a single peer of the overlay networks has to be known by the client.
If those messages are sent to often, peers may terminate connections with the sender.
Or a random one if no peer exists in the same AS as the client requesting the Oracle's assistance.
For the simulations with the different hostcache sizes it is noteworthy to mention that the complete hostcache was always sent to the Oracle. The Oracle then sorted the complete list by the hop distance each IP in the list had in respect to the requesting client to allow localized peer connections. Thus the heuristic used by the Oracle is a simple hop distance based comparison: the further a peer is away from the requesting client, the lower it will be ranked and thus a connection with such a peer will become unlikely.

The results of the different simulations can be classified into the following categories:

**Graph connectivity**  It was verified that the use of the Oracle service did not cause the overlay network to disintegrate into multiple overlay networks and thus does not affect the connectivity of the Gnutella network.

**Mean node degree**  Due to the hierarchical design with normal and ultra-peers it was verified that the use of the Oracle did not impact the number of connections a peer has too much. The mean number of connection per peer stayed in the expected range and thus the Oracle does not influence the overlay network structure.

**Graph diameter**  In the unbiased simulation the graph diameter was 5 - 7 hops. In the case with 100 IPs the diameter increased to 6 - 8 hops and thus is negligible. The average diameter with 1000 IPs per Oracle request increased to 9.2.

**Mean overlay path**  The average path-length in the Gnutella overlay network did change but not significantly.

**Mean AS distance**  The average number of ASes between two peers vastly decreased during the simulations where the Oracle was used with the average dropping below 1, showing the most of the traffic indeed was kept local with the help of the Oracle.

**Intra-AS connections**  The number of connections between peers within the same AS also increased by quite a huge margin, supporting the claim that the Oracle is able to localize the traffic of the Gnutella overlay network.

**Scalability of Gnutella**  Due to the flooding nature of the Gnutella protocol, the oracle was also able to increase the stability of the Gnutella network by reducing the absolute number of Ping, Pong, Query and QueryHit messages to nearly half in the case of an IP list size of 1000.

**Localization of content exchange**  The actual data exchange between peers in the Gnutella network is done via direct HTTP connections. When a peer received multiple nodes offering the requested content it again contacted the Oracle for assistance in selecting a peer. With the help of the Oracle the number of file transfers that stayed within the same AS was increased by 34%. An amount that would have otherwise been downloaded from a peer outside of the querying peer's AS, thus reducing the amount of data flowing through the peering links.

**Download time**  For the end-users the Oracle also has some serious advantages. The simulations haven shown that the average time a user has to wait until the download of a file finishes decreases by 16%-34% in the different topology instances.
Query responses Another gain for the end-user is the average time it takes for a search query to be answered. Due to the improvements in the overlay the Oracle is able to achieve the average time it takes for a search query to be answered is lowered up to 16% [5].

The work of Anja Feldmann and Vinay Aggarwal [2] has shown the impact the Oracle service can have on the Gnutella network and the different gains the Oracle can achieve. One goal of this thesis is to extend those observations to the BitTorrent [13] filesharing protocol, a Peer-to-Peer filesharing protocol that has gained a lot of attention by researchers as well as endusers during the last couple of years [41, 39, 8, 35, 26]. Additionally the impact on different ranking functions for the IP lists by the Oracle will be examined, from the pure hop distance based approach of the Gnutella implementation introduced in this chapter to a more sophisticated one, possibly using multiple different attributes to allow a better suited answer for the different characteristics of a different protocol.
4 The BitTorrent Protocol

In this chapter will describe BitTorrent, a Peer-to-Peer (P2P) file sharing protocol, designed by Bram Cohen in 2001. BitTorrent generally consist of the following entities: a (encoded) metainfo file (a so called "torrent" file), a tracker and the BitTorrent clients. The metainfo file describes the content being downloaded and provides basic download corruption protection by supplying SHA-1 hash values for the files being downloaded. The tracker coordinates the file distribution of one or more .torrent files and the peers participating in downloading the contents of such an .torrent file. All peers downloading the same content and thus being interconnected are called a swarm. All peers a client is interconnected with are called the "local swarm" of that peer. Note that there is a small possibility for peers building distinct swarms although downloading the same content. A peer that has finished downloading the complete content but still participates in the swarm is called a seed. A peer that has not finished downloading the content is called a leecher.

4.1 Bencoding

The metainfo files uses bencoding for data representation and thus avoids problems with byte ordering ("endianness") across different platforms/architectures but is less efficient than a pure binary encoding because of the ASCII encoding used. Bencoding supports four different basic data types

1. byte strings
2. integers
3. lists
4. dictionaries (associative arrays)

and uses ASCII characters as delimiters and digits. The encoding is as follows:

- An integer is encoded as i<number in base 10 notation>e. Negative values are allowed, leading zeros are not.
- A byte string is encoded as <length>:<contents>. The length is encoded in base 10 and must be non-negative. The content is a sequence of bytes that are not necessarily made up of characters. Note that only ASCII encoding is supported, other encodings like UTF-8 are not standardized in the BitTorrent draft.
- A list of values is encoded as l<content>e. The values of the list are concatenated bencoded values without separators between elements.
- A dictionary is encoded as d<contents>e. The elements of the dictionary are again bencoded, each value immediately following the key it is associated with. All keys must be byte strings and must appear in lexicographical order.

4.2 The Metainfo (.torrent) File

The metainfo file, a .torrent file, contains all necessary information about the coordinating tracker and the files being downloaded. The tracker and its functionality is described in detail in 4.3. In BitTorrent a file is subdivided into pieces of a fixed length, only the last piece being irregular. For each of those pieces the corresponding SHA-1 hash is calculated and stored in the .torrent file, which allows a client to notice possible file corruption. Also the tracker managing the peers of this swarm is stored, although it is possible to have multiple trackers in a .torrent file.

The metainfo file is an bencoded dictionary with the following keys (only required keys are listed, for a full reference see [9]):
• info: a dictionary that describes the file(s) of the torrent. There are two possible forms: one for the case of a 'single-file' torrent with no directory structure, and one for the case of a 'multi-file' torrent.

• announce: A string that denotes the announced URL of the tracker

The info dictionary has the following key/value pairs:

• piece length: an integer denoting the piece-size in bytes

• pieces: a byte-string consisting of the concatenation of all SHA-1 hash values, one per piece

and in the case of an 'single-file' torrent it also contains the following keys:

• name: the filename as a byte-string.

• length: the length of the file in bytes encoded as integer.

In the case of an 'multi-file' torrent, it instead contains the keys:

• name: a string that denotes the directory name in which all the files should be stored.

• files: a list of dictionaries, one for each file. Each dictionary in this list contains the following keys:
  - length: length of the file in bytes as integer
  - path: a list containing one or more string elements that together represent the path and in case of the last element the filename.

4.3 The Tracker

The tracker coordinates the file distribution of different .torrent files and the peers associated with them. Whenever a peer wants to join the swarm of a .torrent file it contacts one of the trackers noted in the .torrent file. The reply will then contain a list of peers in that swarm. The tracker is also contacted at regular intervals specified by the tracker itself. The tracker itself is an HTTP(S) service which replies to basic HTTP-GET request. Those HTTP-GET requests have the following fields (only required fields are listed, for a full reference see [11]):

• info_hash: An urlencoded 20-byte SHA1 hash of the value of the info key from the metainfo file.

• peer_id: urlencoded 20-byte string used as a unique ID for the client, generated by the client at startup. This is allowed to be any value, and may be binary data.

• port: The port number(s) that the client is listening on, typically 6881-6889.

• uploaded: The total amount of uploaded Bytes in base ten ASCII.

• downloaded: The total amount of downloaded Bytes in base ten ASCII.

• left: The number of bytes this client still has to download, encoded in base ten ASCII.

• compact: Setting this to 1 indicates that the client accepts a compact response. The peers list is replaced by a peers string with 6 bytes per peer. The first four bytes are the host (in network byte order), the last two bytes are the port (again in network byte order).

• no_peer_id: Indicates that the tracker can omit peer id field in peers dictionary. This option is ignored if compact is enabled.

• event: If specified, must be one of started, completed, stopped. (or empty which is the same as not being specified). If not specified, then this request is one performed at regular intervals.
– started: The first request to the tracker must include the event key with this value.
– stopped: Must be sent to the tracker if the client is shutting down gracefully.
– completed: Must be sent to the tracker when the download completes.

The reply from the tracker is a standard plaintext document (MIME-type “text/plain”) containing nothing but a bencoded dictionary with the following keys:

- failure reason: a human readable string stating the reason a request failed. If this key is present, no other keys may be present.
- interval: an integer denoting the interval in seconds a client should wait between periodical requests
- tracker id: A string that the client should send back on its next announcements. If the tracker id is absent and a previous reply contained a tracker id, keep using the old value. Note that while this field is mandatory in the reply it is optional in the request.
- complete: number of peers with the entire file (seeder)
- incomplete: number of non-seeder peers (leecher)
- peers: The value is a list of dictionaries, each with the following keys:
  - peer id: a string with the peer’s self-selected ID, as described above for the request
  - ip: a string with the peer’s IP address either IPv6 (hexadecimal notation) or IPv4 (dotted quad notation) or DNS name
  - port: an integer denoting the peer’s port number

In case of a compact reply the bencoded list of dictionaries containing the peer information is replaced by a single byte-string with its length being a multiple of 6: the first 4 bytes being the IP address and the last 2 bytes the port number.

4.4 The Peerwire Protocol

The peerwire protocol is used by the peers to exchange the data described in the metainfo file. Each client must maintain the following state it has with its remote peer:

- choked: Whether or not the remote peer has choked the client. Choked means that no requests will be answered or accepted. In this state the client should not attempt to send request and consider all pending (requested but unanswered) requests being discarded by the remote peer.

- interested: Whether or not the remote peer is interested in pieces the client has to offer. This is a notification that the remote peer will begin requesting blocks when the client unchokes them.

This implies that a client also has to track the state itself has with the remote peer, so both states exist twice, once for the state of the remote peer and once for the client itself. Each connection with a peer starts as “choked” and “not interested”. It is important for a client to keep its peers informed of its interested state in a timely fashion, even when being choked, to avoid wasting bandwidth due to being unchoked but uninterested. The client starts requesting blocks when it is unchoked by the remote peer, which requires the client to be previously interested. Vice versa a client uploads a block when he unchokes an interested client and has requests pending.

As described in section 4.2, the basis for sharing data are pieces of fixed length as described in the torrent file. Transmission however is not done on a pure piece-wise basis, but on the basis of blocks. A block is described by the triple (piece number, offset, length), where piece number is the zero-based number of the piece of which a block is requested, offset denotes the zero-based offset in bytes of a piece where the block starts and length is the size of the block in bytes. Although the blocksize is arbitrary, the mainline client and most other clients do use a blocksize of $2^{14}$ byte = 16 KByte and discard requests of bigger blocks.
4.4.1 data-types and message flow

The peerwire protocol consists of a set of length prefixed messages and a handshake message. Unless specified otherwise, integer values are encoded as four byte big-endian. This includes the length prefix on all messages except the handshake. The peerwire protocol starts with the exchange of a handshake message, which is strictly required. The communication onwards consists of exchanging length prefixed messages. The Handshake message is the first message exchanged after the TCP-connection is established and is mandatory. It has a size of \((49 + \text{len(pstr)})\) bytes and has the following format:

\[<\text{pstrlen}><\text{pstr}><\text{reserved}><\text{info}\_hash><\text{peer}\_id>\]

with the fields being:

- \(<\text{pstrlen}>\): string length of \(<\text{pstr}>\), as a single raw byte
- \(<\text{pstr}>\): string identifier of the protocol
- \(<\text{reserved}>\): eight (8) reserved bytes. All current implementations use all zeroes. Each bit in these bytes can be used to change the behavior of the protocol. An email from Bram suggests that trailing bits should be used first, so that leading bits may be used to change the meaning of trailing bits.
- \(<\text{info}\_\text{hash}>\): 20-byte SHA1 hash of the info key in the metainfo file. This is the same info_hash that is transmitted in tracker requests.
- \(<\text{peer}\_\text{id}>\): 20-byte string used as a unique ID for the client. This is usually the same peer_id that is transmitted in tracker requests

In version 1.0 of the BitTorrent protocol, pstrlen = 19, and pstr = “BitTorrent protocol”. If a client receives a handshake with an info_hash it is currently not serving, it must drop the connection. The recipient may wait until it gets the handshake from the initiator if it serves multiple torrents. If the initiator of a connection gets a handshake with the peer_id not matching, the initiator should drop the connection. Remember that the peer_id information is supplied by the tracker with each returned client if not in compact mode (see chapter 4.3).

4.4.2 messages

The remaining messages of the peerwire protocol have the same layout:

\[<\text{length prefix}><\text{message ID}><\text{payload}>\]

The length prefix is a four byte big-endian integer value and is the message-length without those four bytes. The message ID is a single decimal byte. The payload depends on the message type.

**keep-alive:** \(<\text{len}=0000>\) The keep-alive message has the length-prefix set to zero and does not contain a message ID or a payload. Its length is therefore 4 bytes in total. In general a peer may close the connection if no messages have been exchanged for a certain period of time, generally two minutes. The keep-alive message is used to maintain a connection if there are no other messages to be sent during that timeframe and thus prevents the peer from closing the connection due to time-outs.

**choke:** \(<\text{len}=0001><\text{id}=0>\) The choke message consist of the length-prefix and ID field only, resulting in a 5 byte message. The ID is set to 0 and the length prefix is set to 1, due to the length prefix not counting itself to the message size.

**unchoke:** \(<\text{len}=0001><\text{id}=1>\) The unchoke message is exactly structured like the choke message, the only difference being the ID set to 1.
interested: $<\text{len}=0001><\text{id}=2>$ The interested message is a fixed length message like choke and un choke, the ID set to 2. It is 5 bytes in total.

not interested: $<\text{len}=0001><\text{id}=3>$ Like the three messages before, choke, un choke and interested, the not interested message has a length of 5 bytes and the ID set to 3.

have: $<\text{len}=0005><\text{id}=4><\text{piece index}>$ The have message is also a fixed length message with a total length of 5 bytes. The payload denotes the zero-based index of a piece that was downloaded and verified by the SHA-1 hash of the sender of this message. The ID of this message is 4. Note that this message is not mandatory. If a peer already has the piece, this message might be skipped. This “have suppression” called technique saves up to 25% of overall protocol overhead [10].

bitfield: $<\text{len}=0001+X><\text{id}=5><\text{bitfield}>$ The bitfield message is only allowed to be sent immediately after the handshake sequence is completed, before any other messages. This is an optional message and may be skipped if a peer has no or very few pieces\(^8\). The payload of the bitfield message is a bitfield denoting already present pieces of the peer sending this message and is padded to a full byte, where unused bits are set to zero. A bit set to 1 denotes a piece being present, whereas a 0 denotes a missing piece. A bitfield of wrong (padded) length is considered an error and the client should drop the connection. The length of the bitfield message is $5 + \left(\lceil\frac{\text{bitfield length}}{8}\rceil\right)$ bytes.

request: $<\text{len}=0013><\text{id}=6><\text{index}><\text{begin}><\text{length}>$ The request message is used to request a block. The ID of this message is 6. The payload contains the following fields: the index field denotes the zero-based index of the piece of which a block should be downloaded, begin specifies the zero-based offset in bytes where the block starts and length the length of the block in bytes. This message has a fixed length of 17 bytes. For further information about pieces and blocks, see [35].

piece: $<\text{len}=0009+X><\text{id}=7><\text{index}><\text{begin}><\text{block}>$ A piece message is used to transfer the actual content of a requested block. The fields index and begin in the payload match the fields from the corresponding request. It has a variable length depending on the blocks length. It is $13+\text{block length}$ bytes.

cancel: $<\text{len}=0013><\text{id}=8><\text{index}><\text{begin}><\text{length}>$ A cancel message is used to cancel yet unanswered request. The fields in the payload match exactly the fields in the corresponding request message. It is a fixed size message with a length of 17 bytes.

port: $<\text{len}=0003><\text{id}=9><\text{listen-port}>$ The port message is used to denote the listening port of the DHT of a client if the client supports this feature. This feature is also called “DHT Tracker” or “Trackerless BitTorrent”, for further information see [18].

4.5 Choke and Unchoke Algorithms

When a connection between two peers is established, it starts out in choked state, which means that no requests should be made and thus no download of blocks is possible. Changing this state from choked to unchoked is done based on an algorithm which is executed every ten seconds, a so called round. This algorithm is different for peers that are seeders and those that are leechers. The number of parallel unchoked connections is $n$, but note that in the mainline client at most 4 peers are unchoked at any given time.

For Leechers, every time a new round starts the following steps are taken:

\(^8\)In the case of very few pieces, it might save some bandwidth by advertising those pieces via have messages
1. All interested peers are ordered according to their upload rate to the client, ignoring peers not having uploaded in the last 30 seconds.

2. The \( n - 1 \) peers with the highest upload rate are unchoked.

3. In every third round a randomly chosen interested peer is unchoked. This is the “optimistic unchoke”, which allows new peers to join the swarm and gain pieces in order to be able to upload to other peers.

For seeders, the steps taken are:

1. The algorithm first sorts all interested peers based on their last unchoke time, most recently unchoked peers first. Note that this time is the time of the initial unchoke message, when a peer is allowed to download for more than one consecutive timeslot of ten seconds, no additional unchoke message is sent. In case of a tie, the peers are sorted by their download speeds from the seed, fastest peers first.

2. For two consecutive rounds the first \( n - 1 \) peers are kept unchoked and an additional \( n^{th} \) randomly chosen peers is unchoked as the “optimistic unchoke”.

3. In the third round all \( n \) unchoked peers are kept unchoked.

This choking algorithm provides two basic properties. In case of the seed all interested peers are given the equal amount of time to download data and thus allows fast distribution of rare pieces in the peers respective local swarm. In case of a leecher the choking algorithm heavily favors those peers who are uploading data to the client and thus allows for a “tit for tat”-ish behaviour. In both cases the optimistic unchoke allows new peers in the swarm to get initial data and therefore effectively integrating them into the swarm enabling them to get some pieces to offer. In the leecher’s case the optimistic unchoke also allows the client to learn of possible better connections it would otherwise not know of. For further information on the choking algorithm and its properties the interested reader might have a look at [35].

### 4.6 Piece Downloading Strategy

A crucial part in BitTorrent is the selection of pieces which a client wants to download, because of the need to upload data to other peers in order to be able to download from those peers. So the pieces a client wants to have should be the ones that few others have. This heuristic is called the “rarest first” algorithm. Because of the initial bitfield and later have messages, a client can keep track of the pieces its neighbours have that it itself does not have. With this information the client can request the least available pieces first and after receiving and advertising them to other peers will be able to upload them, because only few others already have them, while everyone else wants them. Note that this strategy also focuses on distributing every single piece in a manner that all available pieces have a nearly equal count in the neighbourhood of a peer. Although this approach only uses the information of the local swarm of a peer, it achieves a close behaviour of a rarest first algorithm executed on the global swarm [35].

When approaching the last pieces of a torrent, the client changes this piece selection strategy into one called “endgame”. The endgame strategy requests every block left from all peers that have the respective piece. This means that blocks are requested multiple times from multiple peers and thus might be downloaded more than once. This is done to avoid being unchoked by peers but not requesting any blocks because all block left are already requested from other peers. Thus a client wastes some bandwidth to increase the likelihood of receiving a block as fast as possible to finish the download. Again, for further information and theoretical background, see [35].
4.7 Differences to Gnutella

There are a few but significant differences between the Gnutella and the BitTorrent protocol. One difference is that BitTorrent does not perform any search queries. Thus one major part of the whole downloading process is done “out of band” as the search for content is already finished when someone download a .torrent file because the tracker has all necessary information about participating peers.

The most important difference between BitTorrent and Gnutella is the way a download of content is performed. In Gnutella a single peer is chosen and the download starts from a single source. Thus the impact on download speed of this single source is quite huge, the better this single source performs, the better the achieved performance is. In BitTorrent the content is split into several pieces which can be downloaded in parallel from multiple sources. Thus the impact of a single source is dependant on the number of active connections, the more connections a BitTorrent user can download pieces from, the less the impact of a single connection is to the overall download speed achieved. Additionally the choking algorithm dictates that a user has to upload data to a peer in order to download data, while in Gnutella this is not the case. Recent work has shown that peers with similar upload characteristics tend to build clusters, so peers tend to unchoke peers with the same upload capacities a priori.

Another important fact is that the Oracle can only influence the connections a BitTorrent peer initiates, while there are also incoming connections from other peers which the Oracle has no possibility to alter if the peer is in a foreign network. Again due to the protocol requiring a peer to upload in order to be able to download, this means that only roughly 50% of all connections can be influenced by the Oracle and thus only one half of the overlay network is subject to optimization by the Oracle. As all connections will most likely carry traffic in both directions, only those connections build with the assistance of the Oracle will reduce the possible traffic on peering links.

With this information it is possible to estimate the impact the Oracle can have on the BitTorrent overlay network: Users should be able to achieve higher average download speeds and thus decreased download times, as the created overlay network can be optimized to avoid possible bottlenecks and allows a better peer matching. The amount of peering traffic will most likely be reduced as many peer connections can be altered to stay inside of a network, but due to the nature of the BitTorrent protocol all connections between peers will carry traffic in both directions, the only exception is if one of the peers is a seed.

4.8 Use of the Oracle

The BitTorrent protocol has few locations where one could contact the Oracle and make use of the information it provides. The most obvious location would be just after receiving the peer list from the tracker. A small drawback at this point is that the client has no information if a peer from that list actually has the content it is interested in, as this information is exchanged right after the handshake of the peerwire protocol. Another possibility to contact the Oracle is by the tracker itself. Instead of randomly selecting a certain number of peers from a swarm, the tracker could send the complete IP list of a swarm to the Oracle, thus selecting the best peers for the client from the complete swarm, instead of selecting the best peers from a random subset. As the tracker also has a rough estimate of which peers have how much of the content, one could imagine the use of that information to further refine the list of possible peers by providing this information to the Oracle. As of now there is no possibility to provide such an additional parameter to the Oracle, but the specification of the Oracle allows for quite a few possibilities to add such a capability, such as adding the parameter as an option field per IP or even a new category in the Oracle protocol.

---

9Although recent development in the BitTorrent community has turned to “trackerless” BitTorrent, where a DHT inurs the function of a tracker and does allow for searches based on the unique torrent hashes.
5 Simulation Environment and other used Tools

The simulations conducted in this thesis were done with “SSF Network Models (SSFNet)” [16]. Because no implementation of the BitTorrent protocol were available at the time this thesis began, my own implementation of the BitTorrent protocol on top of SSFNet as well as the Oracle protocol will be shown. Besides the simulation environment various scripts were used to simplify the management of the simulation configuration, automation of multiple simulation runs and evaluation of simulation results. The following chapter will introduce all software components and the tools used to develop them, as well as a short introduction on how to use this environment to run own simulations and get their respective evaluations.

5.1 SSFNet Simulation Environment

The “SSF Network Models (SSFNet)” [16], which was used to conduct the various simulations for this thesis, uses the Raceway “Scalable Simulation Framework (SSF)” [14] Kernel as its basis. SSF is a public domain standard for discrete event simulation and its implementation is available in different languages like Java or C++. The version driving the publically available SSFNet simulator from Renesys Corp. is written in Java and is free for scientific research. The SSF API is available for Java and C++ [51, 52].

SSFNet itself is a suite of open source Java models of protocols like IP, TCP, UDP, BGP4, OSPF, network elements like hosts, routers, links, LANs, and support classes for realistic multi-protocol, multi-domain Internet modeling and simulation [16]. It is a packet level simulator, which means the complete internet protocol stack, including the IP layer, is completely simulated. SSFNet allows for self-configuring models, each SSFNet class instantiation can configure itself by querying a special database written in Domain Modeling Language (DML).

The coding of the lower layers of the IP stack are thus provided by SSFNet, while the BitTorrent protocol as well as the Oracle protocol are implemented as SSFNet applications on top of it. The BitTorrent protocol is subdivided into the client and the tracker application, while the Oracle protocol is implemented as a standalone server application, the client part of the protocol is directly integrated into the BitTorrent client.

5.2 Domain Modeling Language

“DML” stands for Domain Modeling Language and is a public domain standard for attribute databases for model configuration and verification and supports extensibility, inheritance as well as substitution of attributes via special keywords. For detailed information and a formal definition of DML see [15]. A DML expression is a list of whitespace separated key-value pairs, which form a hierarchical scheme. A key is an alphanumeric string and a value is an alphanumeric string or a DML expression. Put together they build an attribute tree with keys as internal nodes and values as leaves. In DML a keypath uniquely identifies an attribute in such a tree by concatenating all keys with a . along the path from the root to the attribute. The reserved words in DML are:

- _find <keypath> is substituted by the attribute specified by the keypath.
- _extends <keypath> is an inheritance keyword. The attribute inherits all key-value pairs of the attribute specified by keypath.
- _schema <keypath> points to another DML tuple which contains a schema definition for the enclosing DML. A schema definition is used to specify concrete DML attribute tree by enforcing the key-names, value types and ranges and the allowed number of attribute instances. For further information on schemas see [15].

Listing 1 shows a very basic DML file.
client_options [
  seeder [
    debug_client false
    online_time 10000
    online_window 50
    port 6881
    max_con 60
  ]
  leecher [
    debug_client false
    online_time 10000
    online_window 50
    max_con 60
    _find .client_options.seeder.port
  ]
  leecher2 [
    _extends .client_options.leecher
  ]
]

Listing 1: "An example DML file"

5.3 BitTorrent-Tracker Implementation

The implementation of the BitTorrent tracker is done in the java package SSF.OS.bittorrent.tracker and consists of 3 classes representing the tracker itself and 2 classes modelling the request and reply messages of the tracker protocol. A tracker extends a "protocol session" which allows an instance of the tracker running on a host inside of SSFNet. Each tracker has a list of "TorrentInfo" object, which represents the served torrent files. Each of those "TorrentInfo" objects then contain "ClientInfo" objects representing the BitTorrent clients participating in the download. Whenever a client connects to the tracker and transmits a "getRequest" message, the appropriate "TorrentInfo" object is located and the user is added. After that the tracker replies with a "getReply" message containing the requested number of randomly chosen peers from the list of clients in the "TorrentInfo" object. A class- and flow-diagram of the tracker implementation can be seen in Figure 4.

![Class diagram of the BitTorrent tracker SSNet implementation](image1)

![Flow diagram of the client tracker communication](image2)

Figure 4: The BitTorrent tracker implementation in SSFNet

The configuration of a tracker for the simulation is done via DML:
ProtocolSession {
  name      server
  use       SSF.OS.bittorrent.tracker.Tracker
  port      10  # server's well known port
  client_time_out 60  # time before a client is considered dead
  debug     false  # enable/disable debugging
  connect_interval 900  # interval between client scrapes
  torrent [
    info_hash  myTorrent  # unique torrent name aka info_hash
    size      1000000000  # size of the torrent download in Byte
    piece_size 256  # piece size of this torrent in KB
  ]
}

Listing 2: "DML configuration of the BitTorrent tracker"

The port specifies the TCP port used by this tracker, the client_time_out specifies the time in seconds before a client is removed from the list of participation peers for a torrent when no messages arrive. Debug is a flag denoting if debug messages should be written into a file tracker_<ip>.txt, where <ip> is the actual IP address of the tracker in the simulation. Additionally, a tracker needs one or more torrent sections, each configuring a separate torrent served by the tracker.

The info_hash value has to be unique for the purpose of distinguishing between different torrents, just like the info_hash in a real torrent file (see section 4.2). Size specifies the size of the file that is downloaded with this torrent. Note that this implementation supports only single file torrents. Piece_size is the size of a single piece for that torrent and thus specifies the number of pieces for that torrent: $\lceil \frac{\text{size}}{\text{piece_size}} \rceil$. Note that instead of having a last irregular piece, the size is increased to match the number of pieces.

5.4 BitTorrent-Client Implementation

The client implementation of the BitTorrent protocol is done in the java package SSF.OS.bittorrent.client and consists of nine classes for the client itself as well as thirteen classes for the peer wire protocol messages, each class being one of the messages mentioned in section 4.4.2 and a common super class for convenient programming. The class diagram of the BitTorrent client can be seen in Figure 5. The design of the BitTorrent client was based on generalized observation in real world implementations of the BitTorrent protocol: the SourceForge [49] project “Java BitTorrent API” [31]. The biggest differences are due to the nature of SSFNet and the way data is transferred in SSFNet. SSFNet does not actually send data but uses references to objects holding the data and calculates the time the transfer takes by the specified size of the data transfer. Thus there can not be any data corruption and there is no need to serialize data or conversion due to endianess. So many network related things like encodings and checksumming via SHA-1 were dropped in favour of a simplified software model.

The main class "BitTorrentClient" is instantiated once per host and resembles the client application running on that host. It has one or more "TorrentSession" which act as the controlling instance for exactly one torrent. This class coordinates all actions needed for this torrent, such as the tracker communication, optionally the Oracle communication and all decision needed like the rarest first piece selection strategy, the initiation of more connections or the refusal of incoming connections if a configurable maximum is hit.

"torrentfile" is a class representing the actual file that is downloaded along with informations about this download. It holds the information about blocks and pieces that were already downloaded, are requested or are still missing. It also accumulates the information about the pieces all remote peers have, thus building the information base for the piece selection strategy.

"ClientSession" is the class resembling the connection between the client and a peer. It holds the socket, the message-queue and the request-queue for that peer connection. "ClientInfo" holds all the information about the peer, which pieces it has, if it is a seed, the TCP/IP information and the state of the peer (interested and choked, see section 4.4).
“TrackerSession” represents the connection between the client and a tracker and exists on a per torrent basis. Each “TorrentSession” has exactly one “TrackerSession” which holds the socket and does all the communication between the client and the tracker for that specific torrent, thus it is the implementation of the client tracker protocol of BitTorrent, see chapter 4.3. “TrackerInfo” holds the IP and port of the tracker as a convenient data class.

“OracleSession” and “OracleInfo” represent the exact same information but for the Oracle server. For the Oracle protocol see chapter 2.3.

Figure 5: Class diagram for the SSFNet implementation of the BitTorrent client

Like most protocols the BitTorrent client is configured via DML. There are a few debug flags to enable verbose diagnostic messages. All tags inside of <> will be replaced by its appropriate values as noted in table 1. Although the BitTorrent client has quite a lot of different configuration options, all of them are easy to understand and in fact shareable (via _find keyword, described in 5.2) between the tracker, seeds and leechers participating in the same swarm of a torrent.

<table>
<thead>
<tr>
<th>tag</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;local IP&gt;</td>
<td>IP of the BitTorrent client</td>
</tr>
<tr>
<td>&lt;remote IP&gt;</td>
<td>IP of the remote host</td>
</tr>
<tr>
<td>&lt;remote port&gt;</td>
<td>port of the remote host</td>
</tr>
<tr>
<td>&lt;info_hash&gt;</td>
<td>unique info_hash of the configured torrent</td>
</tr>
</tbody>
</table>

Table 1: tags for debug filenames

debug_client prints general information about the client in the file client_<local IP>.txt, such as all incoming connections and the torrent requested by this connection as well as global configuration infos such as online time and the time each configured torrent will be started.

debug_torrent enables detailed logging about each configured torrent, such as incoming connections for this torrent as well as received blocks and pieces, timed out clients and infos about the performed choking algorithm. This debugging information is saved to the file client_<info_hash>_<local IP>.txt.

debug_client_con will enable extremely detailed information about each peer connection this client has, each connection logged to a file called <local IP>_<remote IP>_<remote port>.txt. Those files contain all received and sent peer wire messages and request made by the client. Remember that each connection will be logged to a separate file, resulting in quite a lot debugging logs for this client.
debug_client_tracker enables debugging information for the connection between the client and the tracker. Information about the time the tracker is contacted as well as the received peer list will be logged to the file \texttt{client_tracker_<local IP>.txt}.

debug_client_oracle does the same for the messages sent and received by the Oracle. It will log the times that the Oracle is contacted and the list of IPs sent and received to a file \texttt{client_oracle_<local IP>.txt}.

\texttt{online\_time} and \texttt{online\_window} specify the amount of seconds the client will stay online. A uniform distribution will generate a random number in the interval \([\texttt{online\_time}, (\texttt{online\_time} + \texttt{online\_window})]\) which is the time in seconds this client will stay online.

\texttt{port} specifies the TCP port this BitTorrent is listening on for incoming connections.

\texttt{max\_con} specifies a global maximum of incoming connections for this BitTorrent client. After reaching this number of incoming connections, no further incoming connections are accepted, in fact, the client stops listening until the number drops below this threshold again.

\texttt{torrent} is the part which defines the torrent that is downloaded and may be present multiple times. It contains the following values:

\texttt{info\_hash} defines the unique identifier of a torrent file, just like the info\_hash in section 4.2.

\texttt{size} specifies the size of the file that is downloaded with this torrent. Note that this implementation supports only single file torrents.

\texttt{piece\_size} is the size of a single piece for the torrent and thus specifies the number of pieces for that torrent: \[ \left\lceil \frac{\texttt{size}}{\texttt{piece\_size}} \right\rceil \]. Note that instead of having a last irregular piece, the size is increased to match the number of pieces.

\texttt{percentage} is a floating point value denoting the percentage of the torrent already present for this client. This is calculated on a piece by piece basis thus the number of present pieces is \( \left\lceil \frac{\texttt{size}}{\texttt{piece\_size}} \right\rceil \times \frac{100}{\texttt{percentage}} \), which are randomly chosen among all possible pieces with a uniform distribution.

\texttt{start\_time} and \texttt{start\_window} specify the number of seconds the client will stay online for this torrent. A uniform distribution will generate a random number in the interval \([\texttt{start\_time}, (\texttt{start\_time} + \texttt{start\_window})]\), which is the time in seconds in which this client will try to download this torrent.

\texttt{seed\_time} and \texttt{seed\_window} specify the time in seconds the client will stay online for this specific torrent after finishing the download and acts as a seed. A uniform distribution will generate a random number in the interval \([\texttt{seed\_time}, (\texttt{seed\_time} + \texttt{seed\_window})]\) which is the time in seconds this client seed this torrent.

\texttt{max\_in\_con} specifies the maximum number of incoming connections for this torrent. \texttt{max\_out\_con} specifies the maximum number of connections the client will establish with other peers for this torrent. Note that \texttt{max\_in\_con} is overridden by \texttt{max\_con} in the client configuration, so that if the number of \texttt{max\_con} connections for the client is reached, no other connections are accepted, even if \texttt{max\_in\_con} is not yet reached for this torrent. This is especially important, if more that one torrent is being downloaded by this client.
**tracker**  specifies the tracker that is used for this torrent, with nhi being its global NHI address (as IP addresses are assigned dynamically when the simulation starts) and port the port the tracker is listening on.

**oracle**  does the same for the Oracle server, again Nhi being the global NHI address and server_port the port of the Oracle server. client_port specifies the UDP port the BitTorrent client is listening on for replies of the Oracle server. If this part is present in the configuration, the BitTorrent client will use the Oracle service, if it is omitted no contact to the Oracle server is made.

```plaintext
ProtocolSession [ 
  use SSF.OS.bittorrent.client.bittorrentClient 
  name client 
  debug_client false # print client debug info 
  debug_torrent false # print debug info for each torrent 
  debug_client_con false # print debug info for each peer connection 
  debug_client_tracker false # print debug info for the tracker connection 
  debug_client_oracle false # print debug info for the oracle connection 
  online_time 10000 # minimum time in seconds the user will stay online even if DLs haven't finished/started 
  online_window 50 # stay online a randomly chosen time in interval [online_time, online_time+online_window] 
  port 6881 # listening port for incoming connections 
  max_con 30 # global maximum number incoming of connections 
]

torrent [ 
  info_hash myTorrent # unique torrent name aka Info_hash 
  size 100000000 # size of the torrent download in Byte 
  piece_size 256 # piece size of this torrent in Byte 
  block_size 16 # block size of this torrent in Byte 
  percentage 0.0 # percentage of the download already finished (rounded to the next full number of pieces) 
  start_time 5.0 # earliest time to send request to the tracker 
  start_window 500.0 # in interval [start_time, start_time+start_window] 
  seed_time 200 # minimum time in seconds the user will seed this torrent (0 = immediate disconnect) 
  seed_window 200 # seed torrent a randomly chosen time in interval [seed_time, seed_time+seed_window] 
  max_in_con 30 # maximum incoming connections for this torrent 
  max_out_con 30 # maximum outgoing connections for this torrent 
]

tracker [ 
  nhi 0:0:200(0) # global Nhi of the tracker 
  port 10 # listening port of the tracker 
]

oracle [ 
  nhi 0:0:300(0) # global Nhi of the oracle server 
  server_port 10 # listening port of the server 
  client_port 20 # listening port for the client 
]
]
Listing 3: "DML configuration of the BitTorrent client"
```
5.5 The Oracle Implementation

The Oracle implementation for SSFNet is straightforward: it uses a single class along with a helper class and a single message and is implemented in the java package SSF.OS.oracle. A class- and flow-diagram can be seen in Figure 6.

Whenever a client contacts the Oracle server an `oracle_message` containing a list of IP addresses is sent to the Oracle via UDP. As a constraint this UDP message can not exceed 1500 bytes and the constructor will cut the number of IP addresses if necessary. The Oracle then collects all possible path information from the client as a source to each IP address in that list as destination by walking the actual SSFNet data structures. The collected information includes:

- The minimum bandwidth of the path in both directions
- The aggregated delay of the path in both directions
- as well as the maximum delay of a segment of the path in both directions
- The hop count
- The AS path

After collecting this information each IP is assigned a weight, represented by the `weighted_ip` class, via a weighting function, where higher weights are preferable peers. Then all IP addresses are sorted by their annotated weight and this sorted list is sent back to the client. Note that the client does not get the weight for an IP address let alone any of the mentioned information. The reply contains only a re-ordered list of IP addresses, where the first one is the most preferable for the client.

Like everything else, the Oracle is also configured via DML and has a few interesting options:

```java
ProtocolSession {
    name server
    use SSF.OS.oracle.Oracle
    port 10 # server's well known port
    debug true # enable/disable debugging
    use_congestion true # whether to use congestion scaling

    # example of link cost annotation:
    # links (or rather network cards) have to be global SHIs
    # link_a -> link_b with weight 5
    # cost for link_b -> link_a has to be specified on its own
```
As always `port` specifies the TCP port used, `debug` allows for debugging messages in the file `oracle_server_<IP>.txt` with `<IP>` being the IP assigned to the Oracle server in the simulation. `use_congestion` is a special flag that allows the use of a utility class called “linkLoad” which monitors the congestion and overall data transfer of a link. When this flag is set to true, the Oracle considers the actual available bandwidth of a link instead of its capacity when gathering the bandwidth information of a path, provided this information is available.

The class `linkLoad` is implemented in `SSF.OS.linkUtil` and has to be configured via DML to gather the wanted information about congestion and transferred data. To enable such a monitoring, an interface of a host in the network has to be annotated with the following DML snippet:

```dml
monitor [ use SSF.OS.linkUtil.linkLoad ]
```

Note that not a link but rather a network interface monitors the congestion level and transmitted amount of data because of SSFNet's restrictions in availability in layer 2 information.

`link_cost` is a feature to allow cost annotations of links. This enables a simple “policy”-like decision for the Oracle. When a link with an annotated cost is found on the path, this cost (or rather the sum of all costs on the path) is also available to the ranking function. To specify a cost for a link `link_a` and `link_b` have to be global NHIs of network interfaces that are interconnected via a point-to-point link. `cost` specifies the value and is an arbitrary number, the higher the cost, the less likely this path will be chosen. Note that the cost for a link is only unidirectional from `link_a` to `link_b`.

### 5.6 Used Tools

The tools used to create the SSFNet implementation of the BitTorrent protocol and the Oracle protocol are Eclipse [19] and the Sun Java SE Development Kit (JDK) 1.6 [40] using the appropriate Interface definitions for a SSFNet protocol implementation. For the simulation framework the tools of choice were simple bash [42] scripts in combination with various tools easing the process of deploying a simulation run. To allow parallel execution of multiple simulation runs GNU make [37] was used as it is able to resolve dependencies, which is needed in case of multiple runs of the same simulation setup for averaging to allow the evaluation scripts running after all repetitions of a simulation were done. To gather the needed data from the available logfiles and supplying them in a usable format for GNU R [23], an analyzing script utilizes GNU awk [1] in combination with GNU grep [21], sort [46] and sed [48]. For each single run of a simulation a GNU Screen [47] session was used to allow better monitoring as well as detaching of long running simulations. For the statistical analysis of the different simulations GNU R was the tool of choice, as it is very flexible and easily scriptable and comes with built-in support of many plot types, as the used box-plots which includes the mean as well as the 75th- and 95th-quintile. Finally GNU tar [56] and GNU zip [22] were used to prevent massive disk space consumption\(^\text{10}\) by archiving old, already analyzed simulations.

### 5.7 Script Framework

To be able to deploy simulations easily, a simple framework was created to easily configure and run the anticipated simulations. The framework consists of a number of bash-scripts:

\(^{10}\) A typical simulation conducted for this thesis created around 5 GBytes of logfiles. For our base-line simulations we had eight different simulation setups resulting in nearly 45 GByte of uncompressed logfiles. Archiving them weighted in around 7 GByte.
run_sim.sh acts as the main entry point for the simulation framework. It merely calls the other scripts and thus allows the chaining of the different steps taken by each script. With correct configuration, a simple `sh run_sim.sh` will trigger a complete simulation framework cycle consisting of: simulation, data analysis, plot generation, archiving. The needed configuration for the framework will be explained on a step per step basis, each step separated into its own script.

config.inc contains commonly used variables and acts as the configuration of the framework. The file contains the following variables:

- `BASE_DIR` defines the directory in which all of the above mentioned scripts reside.
- `EVAL_DIR` is the directory where all GNU R scripts and plots are stored.
- `IGNORE_DIRS` is list of directories which are never considered as a data source in any of the scripts.
- `R_FILE` defines the name of the temporarily created GNU R script file.
- `ANALYSIS_FILES` is a list of BitTorrent info_hashes (see chapter 4.2 for further information) for which the analysis script will generate plots.
- `DATA_FILE`, `PIECE_FILE` and `BLOCK_FILE` are the prefixes for the files which will be created for usage of GNU R.
- `SSFNET_DIR` is the path where the needed .jar archives for SSFNet reside.
- `SSFNET_DML_DIR` is the path where all needed DML files are stored.
- `SSFNET_DML` is a string specifying all DML files used by all simulation.
- `SSFNET_JAR` contains all used jar files when invoking the Java VM.
- `JAVA_MAIN` contains the main class for the VM invocation.
- `JAVA_BIN` contains the commandline argument for the VM.
- `SCREENRC` denotes a .screenrc file to configure the individual screen sessions.
- `SCREEN_BASE` contains all commandline arguments for GNU screen.
- `SIM_DIR` is the folder where all simulation logs will be created.
- `SIM_TIME` is the time in seconds that will be simulated.
- `SIM_RUNS` is the number of repetitions for each simulation.
- `MAKEFILE` is the name of the created Makefile.
- `CLEANUP` is a boolean value specifying if all produced logfiles of this invocation of the framework should be archived.

create_sims.sh is responsible for creating the Makefile (using `MAKEFILE` as filename) for parallel execution and evaluation of all simulations. It looks up all DML files in `SSFNET_DML` that start with "network_" and considers each of them a unique simulation configuration, a so called case. For each case `SIM_RUNS` make targets are created, each invoking a screen session (using `SCREENRC` and `SCREEN_BASE`) with the
complete java VM commandline arguments (using SSFNET_JAR, JAVA_MAIN and JAVA_BIN) starting the simulation. Additionally each case has a make target which depends on the SIM_RUNS executions of the case, executing the data analysis and plot generation as well as the optional (via CLEANUP) archiving. A convenient “all” target is also created, which only depends on all other make targets but does not execute any additional commands, thus invoking the well known “make all” will successively execute all repetitions of all cases, start the data analysis as well as the plot generation.

**analysis.sh** is the script responsible for gathering the needed information from the logfiles produced by each simulation run in a format usable by GNU R. ANALYSIS_FILES specify the unique info_hashes for which the analysis should be done. Thus the analysis is done on a per-torrent basis.

The bash scripts r_avg_dl.sh, r_avg_dl_box.sh, r_con_track.sh and r_traffic_congestion.sh are the scripts responsible for creating the GNU R script files generating the different plots and invoking GNU R. They use several config options: R_FILE specifies the temporary file used to store the R script. ANALYSIS_FILES, DATA_FILE, PIECE_FILE and BLOCK_FILE define the data files used by GNU R. EVAL_DIR is the directory where all the plots as well as the R scripts are stored. All scripts create one plot for two simulations, with and without the usage of the Oracle, to provide a direct comparison between both simulation runs.

**r_avg_dl_box.sh** creates boxplots of the average download speed per BitTorrent client grouped by the different network sizes. The plot contains two plots for the average download speed, each with and without the use of the Oracle:

1. all client
2. all clients of our own network
3. all clients of all Tier-3 networks
4. all clients of all Tier-2 networks
5. all clients of all Tier-1 networks

**r_avg_dl.sh** plots the average download speed over time of a peer. For each network a separate plot exists where two curves are plotted, one for the simulation without the Oracle and one for the simulation where the Oracle was used by the clients in our own network.

**r_traffic_congestion.sh** creates four plots per simulation: two of the peering link congestion (incoming and outgoing) and one each for the total amount of transferred data into and out of the network operating the Oracle server, which is in this case our own network.

**r_con_track.sh** is the script responsible for finding the unique traffic carrying connections mentioned in chapter 7. It utilizes the perl script analyse_connections.pl, a script gathering information about connections between peers within our own network and also connections between peers in our and peers in a foreign network. It will create a table showing the number of connections between a source AS and a destination AS, which is then combined into a single row showing the number of internal, incoming and outgoing connections for our own network only. The resulting plot is a barplot, showing the average number of connections inside our network and also between our and any foreign network for each simulation compared with the same number for the simulation where no Oracle is used. For further details on connection tracking and its meaning see chapter 7.
**cleanup.sh** will archive all logfiles created by this invocation of the simulation framework on a per case basis, such that the logfiles for each case get stored in a separate archive.

Together with proper configuration, all those scripts build an easy to use simulation framework tailored to allow multiple simulations of many different cases without human interaction. One major point when running simulations is free disc-space. The archiving script can only run after all simulation runs of a case have been done. Thus the required disc-space should at least suffice to provide enough free space for all repetitions of a case. For the main cases used in this thesis (see chapter 7) the required space for a single case was around 7 GByte. Remember that the space required for the archive has to be added as well. The evaluation and plotting of simulations is done on a per-case basis. This means that all repetitions of a case are used for the averaged result. The plots are named after cases, so be aware that when you re-run a whole case, the plots will be overriden and all precious simulations results (if not removed from the directory structure) will be used for the averaged result.
6 System Design and Metrics

This chapter will explain the network layout used for the various simulations as well as the metrics used to analyse the results. Starting with a description of how we found and agreed on an appropriate network model for our simulations, it will continue with an explanation of the different factors which we changed during the various experiments. Last but not least the data we took to analyse the simulations and thus which benefit the Oracle can bring in such an environment is explained.

6.1 The Network Model

The basis for all network simulations is the network itself. As the proposed Oracle service aims at Internet Service Providers and their customers (ISPs), we agreed upon following the general network design of major ISPs: creating a (relatively small) number of Points of Presence (PoPs), physical locations aggregating all customers within a geographic area, which in turn are interconnected via high-speed links.

As the network topology is considered highly sensible and critical information by most ISPs, we turned to public available information about network topologies, such as the network maps provided by the Rocket-Fuel project [50], which created network topologies for multiple US american ISPs back in 2002. Figure 7 shows one of the created network topologies on a PoP level for a major US american ISP. We used this network layout as a basis for our own simulated network. Figure 8a shows the simplified network model on a PoP level deducted from the topology provided by the Rocket-Fuel project in Figure 7. Each PoP is considered a separated physical location hosting several routers: a router for interconnecting the PoPs, a BGP router for interconnection with other networks and some access routers for customers. This structure can be seen in Figure 9a. The bandwidth distribution of the customers per PoP can be seen in table 2a, which results in an aggregated upload bandwidth of 17,424 MBit/s and an aggregated download bandwidth of 189 MBit/s per PoP. With 30 users per PoP and 10 PoPs overall our network hosts 300 customers, having 174.24 MBit/s upload capacity and 1890 MBit/s download capacity. This network is assigned the AS11 number 0 and will be the AS which operates the Oracle server and offers the service to its customers. Only customers within this network can use the Oracle service.

To finish the network design, we added nine foreign networks which interconnect with our AS with, to

---

11A Autonomous System (AS) is a group of IP networks under control of a single authority with a clearly defined external routing policy. A Autonomous System Number (ASN) is a globally unique number identifying an AS and is used by routing protocols to exchange routing information between different ASes.
Figure 8: The network layout of the AS operating the Oracle service.
simulate the interaction between networks not using the Oracle and our network using the Oracle. Those nine networks were divided into three different sizes, which follow the widely adopted Tier-1 to Tier-3 terminology: one network is considered a Tier-1 network, hosting roughly the same number of hosts as we do. Two are considered Tier-2 having 50 hosts each while the rest are considered Tier-3 networks with 20 users. To allow better grouping of same sized networks, the AS numbers of those networks are 10-15 for the Tier-3 networks, 20-21 for the Tier-2 networks and 30 for the Tier-1 network. The network design of each of those providers is even more simplified than our own network. This is mainly done to reduce the complexity and thus runtime and memory constraints of the simulation. Instead of having multiple interconnected PoPs hosting the customers, each network has a single PoP with a single router hosting all customers. For interconnecting with other ASes, a BGP router is also present. A Tier-1 network has a single peering with our network, the Tier-2 networks have two peerings, each link to a different PoP in the same geographical area. The Tier-3 network has seven peerings with our network, more links to geographic areas with denser PoP population. The peerings of the different foreign networks with our network can be seen in Figure 8b, the generalized design of the foreign networks can be seen in Figure 9b. The idea behind this design is simple: if a peering with another network exists, it is usually a link between two routers in the same facility, such as one of the famous internet exchange point (IX or IXP) [29] or private co-locations in data-centers, resulting in nearly no delay when a packet traverses from one network into another. Once the data arrives in a network, we consider the delay until it reaches its destination nearly equal for all entry point and destination pairs, because of the typical hierarchical network design. Thus we do not need to model this in such a detail but can estimate the delay as an average to ease the processing demands of our simulation environment.

6.2 The Ranking Function

In this section the ranking function of the Oracle that was used in the experiments is described and the reasons why it was chosen is also given. Additionally, each parameter used in the function is explained, including what is expected of the ranking function.
As described in chapter 2, the Oracle server ranks each IP address with respect to the client and the properties of the path between the client and this IP address. Each IP address is given a weight, an positive integer value denoting how “good” this IP address, or rather the path, is for the client. The problem here is, that “good” is always relative. For different applications, different definitions of a “good” connection exist. For example a streaming application the possible speed of the connection does not matter that much, as long as it suffices. But how low the latency as well as the jitter for that connection is does matter. On the other hand, a download, like we want to simulate with BitTorrent, will certainly want as much bandwidth as possible, but wouldn’t care too much about the latency.

The different factors were introduced in section 5.5. As we’re using BitTorrent as our protocol, we’re interested in fast downloads, which leads to bandwidth as the dominating factor. So we want to match a client with the peers having the highest upload bandwidth. Additionally, we want to localize traffic for three reasons:

1. Peering links often pose a bandwidth bottleneck between two networks
2. Saving peering traffic saves the network operator’s money
3. The ability to “engineer” the overlay network and thus regaining traffic engineering capabilities

Because the AS-distance between two peers implies the number of peering links on the network path, this leads us to the used weighting function: \(\text{weight} = \frac{(\text{peer}_{\text{upload bandwidth}})}{\text{AS}_{\text{distance}}}\). This function weights peers higher the more bandwidth they have but biases them by their AS distance, which means a peer who is twice as far away (from an AS perspective, not a hop perspective) has to offer twice as much bandwidth to get the same weight. Thus we tend to rate peers within our own AS higher unless the peers outside the AS offer more bandwidth to our client. We do not try to use only local peers, but try to first find fast peers and then to keep the traffic within our own network, such that our client should see an increase in download speed, as well as we expect to see a decrease in transferred traffic over our peering links. Additionally we wanted the ranking function to be as simple as possible while achieving our desired properties.

### 6.3 Influential Parameters

The biggest aspect of this thesis is the correct understanding of the influence of different factors on the Oracle and how we can use this understanding to tailor the ranking function to match our needs. One important general observation is that the impact of the same factor might be different for different protocols. For example latency influences the performance of Gnutella pretty heavily, as the search within the Gnutella network will need more time and the download of a file is done via a single TCP connection, thus the bandwidth delay product limits the achievable performance. On the contrary, BitTorrent is hardly influenced by higher latencies, as no search is conducted in the overlay and because the actual download in BitTorrent utilizes a lot of parallel TCP connections, thus the impact of latency in the bandwidth delay product is lowered due to connection pipelining.

So our approach is to first identify the most influential factors for the performance and then create network scenarios where exactly those factors are massively increased in one scenario and nearly non-existent in another one. With those scenarios we’re in the place to actually evaluate impact of the different factors to the performance. With this information we’re then able to tailor an Oracle ranking function, that allows us to circumvent those explicit bottlenecks and increase the end-users performance.

With BitTorrent being our protocol of choice, one big factor is bandwidth, as the end-user is interested in fast downloads. In our network we consider two different bandwidth aspects: as shown in [2] the last hop bandwidth is crucial to the download performance on a per connection base. Thus we want our clients to get peers with high upload capacities. Recent studies [4, 36, 38] showed that bandwidth bottlenecks nowadays also exist inside the provider networks as well as within the peering links of the providers and even have nearly the same likelihood of existence. We decided to only consider the peering links as

---

12This is different if a DHT is used, but this case is not considered in this thesis.
bottlenecks, as intra-network congestion can be dealt with more easily, for instance via route changes. So the two bandwidth factors we are considering are the peering link bandwidths as well as the upload capacities of our peers. Although we see the upload capacity of the peers as a crucial factor, we do not intend to change them during the simulations, but instead will use the Oracle function to consider them for better recommendations. The reason is that BitTorrent is not aware of this property of a peer and with its uniformly random selection of peers can not take any additional properties of peers into account. Thus we need to keep the basic network layout exactly the same for each simulation with and without the recommendations of the Oracle to be able to compare them. While it would be possible to create a network with only low-speed or only high-speed peers, we decided to stick with a mixed peer setup to not further increase the amount of simulations needed for this thesis.

Another big factor of BitTorrent's efficiency is the upload capacity of the initial seed [34]. Instead of modifying just the initial seeds upload capacity, we decided to also change the location of the initial seed. While the impact of underprovisioned seeds is already well understood, we wanted to actually investigate the impact of the location of the initial seed. With the described peering bandwidth bottlenecks described above this poses quite an interesting scenario: if the seed is in a network which we have a small peering link bandwidth with, how does this affect the overall BitTorrent performance? Can the Oracle help in this scenario or even lessen the impact of underprovisioned seeds/networks? Thus we decided two different locations for our initial seed, one is our own network while the other is one of the Tier-3 networks. We chose one of the Tier-3 networks, because of their small number of hosts, so the content spreading is limited to fewer peers in the initial network and thus the possible peering bottleneck induces underprovisioning of the seeds in that network.

6.4 Evaluation Metrics

To allow a detailed analysis of the different cases explained above, one should consider how “performance” should be defined in this case. As for the end-user this is quite a simple task. An end-user is interested in faster downloads resulting in lower downloading times. Thus the average download speed seems a pretty good metric to analyse the performance gains of the end-users. To see how the download speed of the peers in general is affected, boxplots showing the median as well as the 50- and 95-quantile will be supplied for each simulation.

For the ISP this is a bit more complex. As stated in section 6.2, one of the most important things to the network operator is the ability to engineer traffic. Thus the sheer possibility to influence the peers of an overlay network in regards to its peer connections is already a big gain, especially if that influence increases the customer's performance. To measure the level of influence the Oracle can provide, we will give the number of data flows that exist within our own network and the number of data flows that go outside of our network. Another important factor for an ISP is the amount of traffic that flows through its peering links with other networks. To allow for a detailed analysis of the possible savings in peering traffic, the average congestion over time of all peering links separated by incoming and outgoing traffic for each experiment is provided for the network operating the Oracle.

With this four metrics we will be able to answer the questions of and how good the Oracle recommendations can help with improving the performance for both entities, the end-users as well as the network provider.
Table 2: Distribution of up- and download capacities in the different networks

<table>
<thead>
<tr>
<th>Network</th>
<th>Download</th>
<th>Upload</th>
<th>Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>Own network</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 MBit</td>
<td>128 KBit</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2 MBit</td>
<td>192 KBit</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>6 MBit</td>
<td>256 KBit</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16 MBit</td>
<td>512 KBit</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>10 MBit</td>
<td>10 MBit</td>
<td>3 (shared)</td>
<td></td>
</tr>
<tr>
<td>Sum PoPs</td>
<td>1890 MBit</td>
<td>174,24 MBit</td>
<td>300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Network</th>
<th>Download</th>
<th>Upload</th>
<th>Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tier-1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 MBit</td>
<td>128 KBit</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2 MBit</td>
<td>192 KBit</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>6 MBit</td>
<td>256 KBit</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>16 MBit</td>
<td>512 KBit</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>10 MBit</td>
<td>10 MBit</td>
<td>2x3 (shared)</td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td>1208 MBit</td>
<td>70 MBit</td>
<td>250</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Network</th>
<th>Download</th>
<th>Upload</th>
<th>Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tier-2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 MBit</td>
<td>128 KBit</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>2 MBit</td>
<td>192 KBit</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>6 MBit</td>
<td>256 KBit</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>16 MBit</td>
<td>512 KBit</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>10 MBit</td>
<td>10 MBit</td>
<td>3 (shared)</td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td>275 MBit</td>
<td>23 MBit</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Network</th>
<th>Download</th>
<th>Upload</th>
<th>Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tier-3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 MBit</td>
<td>128 KBit</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2 MBit</td>
<td>192 KBit</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>6 MBit</td>
<td>256 KBit</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td>78 MBit</td>
<td>4,3 MBit</td>
<td>20</td>
</tr>
</tbody>
</table>
In this chapter the different conducted experiments and their results will be discussed. As explained in section 6.3, we varied two basic parameters. The first factor is the existence of a peering link bottleneck between all interconnected networks. If such a bottleneck is present, the accumulated capacity of all peering links a network has, is roughly 10% of the combined upload capacity all clients in this network have. If such a peering link bottleneck is not present, the capacity of each peering link is set high enough to allow every possible peer using its complete upload capacity through that link without rising the congestion levels above 10%. All links within a network have more than enough capacity to avoid any intra-network congestion.

To allow a distinction between the ten different networks based on their respective size all peers within AS 10-15 are grouped as Tier-3, all peers within AS 20-21 are grouped as Tier-2 and all peers within AS 30 are grouped as Tier-1. Peers within our own network are shown separately and will not be included in any of the tier aggregated peers. This is important for the second varied factor which is the position of the initial seed. In the first case the initial seed is located within our own network, while in the second case the location of the initial seed is within one of the Tier-1 networks. By combining both factors, we have four different cases:

1. peering bottlenecks and initial seed in our own network
2. peering bottlenecks and initial seed in a Tier-1 network
3. no peering bottlenecks and initial seed in our own network
4. no peering bottlenecks and initial seed in a Tier-1 network

with each case being simulated with and without the use of the Oracle by the peers in our own network. Additionally we launched experiments to see the behaviour of an Oracle aided overlay network in the case of an overprovisioned seed, like in a scenario where a commercial provider adds few but well provisioned sources to the swarm in order to distribute the content faster. So we chose a scenario where the Oracle could not improve the performance by a huge margin and added an overprovisioned seed to the swarm to analyse the impact an overprovisioned seed can have on such a scenario. Again we varied the initial position of the overprovisioned seed to see how the initial position influences our results with the Oracle.

Another aspect simulated concerns the number of torrents a peer downloads in parallel and thus the number of distinct swarms. To see the effects multiple swarms can have on each other, we simulated two different torrents being downloaded by each peers with both seeds being in our own network and peering bottlenecks between all networks being present.

Throughout the different simulations some parameters were kept the same:

- Only peers in our own networks (ASN 0) can use the Oracle service.

- The time frame in which all peers joined the swarm was 300 seconds and uniformly distributed among all peers (a flash crowd scenario).

- The downloaded file was 100 MByte in size, the piece size was set to 256 KByte.

- The time simulated was 10000 seconds if not specified otherwise.

- Each simulation was repeated five times and the results were averaged.

Additionally to the figures about the achieved average download speed per peer and the average congestion level on the incoming and outgoing peering links, each chapter discussing one or more simulations contains a figure with the numbers of unique data-carrying flows that exist between our own network and all foreign networks as well as internal flows, that are data-carrying connections between peers inside of our network. To allow the differentiation and analysis of traffic flows in and out of our own network, a data-carrying connection is always treated as being unidirectional. If a connection carries data in both
directions it is counted as two separate flows. With this approach the numbers of the different types can be used to analyse the flow of traffic and such yields a metric to see the Oracles traffic engineering capabilities.

A flow is classified as follows:

- **An internal flow** is a connection that exists between two peers within our own network and carries at least one block of data.

- **An incoming flow** is a connection between a peer in our network and a peer in a foreign network that carries at least one block of data from the foreign network into our own network.

- **An outgoing flow** is a connection between a peer in our network and a peer in a foreign network that carries at least one block of data from our network into the foreign network.

Before describing how one can deduce the traffic engineering capabilities of the oracle from those numbers a general characterization of a connection between a peer in our own network and a peer in a foreign network will be given to show in which case such a connection will be an incoming or an outgoing flow and when such a connection will be classified as both an incoming and an outgoing flow.

<table>
<thead>
<tr>
<th>simulations</th>
<th>initial seed</th>
<th>oracle</th>
<th>flows</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>position</td>
<td>used</td>
<td>internal</td>
</tr>
<tr>
<td>yes</td>
<td>own network</td>
<td>no</td>
<td>4008.2</td>
</tr>
<tr>
<td>yes</td>
<td>own network</td>
<td>yes</td>
<td>6886.6</td>
</tr>
<tr>
<td>yes</td>
<td>Tier-1</td>
<td>no</td>
<td>3443.75</td>
</tr>
<tr>
<td>yes</td>
<td>Tier-1</td>
<td>yes</td>
<td>4078.75</td>
</tr>
<tr>
<td>no</td>
<td>own network</td>
<td>no</td>
<td>3758.2</td>
</tr>
<tr>
<td>no</td>
<td>own network</td>
<td>yes</td>
<td>6526</td>
</tr>
<tr>
<td>no</td>
<td>Tier-1</td>
<td>no</td>
<td>3612</td>
</tr>
<tr>
<td>no</td>
<td>Tier-1</td>
<td>yes</td>
<td>6330.6</td>
</tr>
<tr>
<td>yes</td>
<td>own network</td>
<td>no</td>
<td>2549.75</td>
</tr>
<tr>
<td>yes</td>
<td>own network</td>
<td>yes</td>
<td>3406.75</td>
</tr>
<tr>
<td>yes</td>
<td>Tier-1 (10 MBit)</td>
<td>no</td>
<td>3203.4</td>
</tr>
<tr>
<td>yes</td>
<td>Tier-1 (10 MBit)</td>
<td>yes</td>
<td>5383.8</td>
</tr>
<tr>
<td>no</td>
<td>Tier-1 (10 MBit)</td>
<td>no</td>
<td>3145.6</td>
</tr>
<tr>
<td>no</td>
<td>Tier-1 (10 MBit)</td>
<td>yes</td>
<td>5628.4</td>
</tr>
<tr>
<td>yes</td>
<td>own network</td>
<td>no</td>
<td>3011</td>
</tr>
<tr>
<td>yes</td>
<td>own network</td>
<td>yes (1 of 2)</td>
<td>3467</td>
</tr>
<tr>
<td>yes</td>
<td>own network</td>
<td>yes</td>
<td>3939</td>
</tr>
</tbody>
</table>

Table 3: Average number of unique traffic flows in the various simulations

When a peer contacts another peer it does so because it wants to download as many blocks of data as possible from that peer. If a remote peer has nothing the connecting client is interested in, the client usually disconnects from that peer and then tries to establish a new connection with a new peer in order to discover additional data sources. While each connection is a peer connection and thus data can flow in both directions, the emphasis on data flow in such a connection is from the remote peer to the client which opened the connection: If the remote peer is a seed, this is quite obvious, as a seed is no more interested in content and only contributes by uploading data so the traffic flows only from the remote peer to the client. If the remote peer is also a leecher the choking algorithm takes the amount of uploaded data into account when deciding which peers are allowed to download from this peer. Thus a client can only download if it also uploads data to the remote peer. In this case the traffic flows in both direction. But the choking algorithm also allows for an optimistic unchoke, where a randomly chosen peer is allowed to
download for a certain time without having to upload any data. While the choking algorithm is the same for both peers, the conditions under which a peering connection is kept slightly favours the data-flow from the remote peer to the client: while the client connecting to the remote peer would close the connection if the remote peer has nothing the client is interested in, the remote peer would not close this incoming connection. Thus there is a slight emphasis on data-flow from the remote peer to the peers that initiated the connection. In combination this leads to an emphasis on data-flowing from the remote peer to the client that initiated the connection that is mainly dependant on the number of seeds in the swarm.

When considering the different classifications of flows, it is noteworthy that the connections carrying data into a foreign network are less likely to be influenced by the Oracle, as those connections are mainly initiated by peers from that foreign network and the Oracle can only influence the initiated connections from the peers in its own network. Thus one should expect that the number of outgoing flows stays roughly the same, while the number of incoming flows drop and the number of internal flows increases with the use of the Oracle.

With the definition and explanation of the different flow types it is now possible to deduce the traffic engineering capabilities of the Oracle. The increase of internal flows is a metric of how many of the traffic carrying peer connections have been successfully localized with the help of the Oracle by advertising peers in our own network while the drop of incoming flows shows how many traffic carrying peer connections could be avoided by this localization of peer connections. The number of the different traffic flows for all simulations discussed in this chapter can be seen in Table 3.
7.1 Case 1: Congested peering links with seed in own network

Before explaining the results of this experiment, the important settings that characterizes this scenario are summed up:

- In this simulation all peering links between all networks pose a bottleneck and have a capacity of roughly 10% of the combined upload capacity of all nodes in the respective network.
- The initial seed is placed in our own network and is a normal client having 16 MBit/s download and 1 MBit/s upload capacity.
- The simulated time was 10000 seconds, which is fairly enough for most of the peers to finish the anticipated download.\textsuperscript{13}
- The torrents file-size is set to 100 MByte and has a piece size of 256 KByte.
- For detailed network related information such as hosts per network, peerings between the networks or distribution of up- and download-capacities in each network see section 6.1.

Figure 10a is a barplot of the achieved average download speed per peer, the blue bars showing the average download speed of all peers, the green bars showing the average download speed of peers in our network and the grey ones showing the average download speed of peers grouped by their respective networks’ size. While the average download speed for all peers (blue) went up by around 13%, the increase in average download speed for peers in our network (green) was 18%, while the Tier-1 network saw an increase of roughly 11%, and both Tier-2 and Tier-3 peers saw a slight increase in download speed of 4-5%. In this case the potential gain for the end-user with the assistance of the Oracle is quite obvious, an average reduction of time needed to download desired content by nearly one fifth can be considered.

\textsuperscript{13}The time needed was actually tested by running the simulation with different simulated time settings and picking the one which had enough peers finished the download. This timeframe then was used for the other settings as well.
tremendous. Even for the peers in foreign networks the optimized overlay network yields some speed improvements in relation to the networks’ size, bigger networks gaining more.

Figure 10b shows the average download speed over time of a peer within our own network, once without the help of the Oracle (red) and once with the Oracles assistance (green). The much steeper increase in download speed shortly after the beginning of the simulation hints at a faster local piece distribution. With the assistance of the Oracle more local peer connections are established and thus the distribution of pieces is sped up because the peering bottleneck is circumvented. At any time during the simulation the performance of a peer was significantly better than in the case where no Oracle was used. Another effect that can be seen in the plots is the BitTorrent endgame which is much more distinct due to the higher availability of localized content. Because a peer requests all remaining pieces from all available peers in the endgame phase of BitTorrent the average download speed decreases due to pieces being downloaded possibly multiple times in parallel. Because more local peers have more pieces with the help of the Oracle this effect gets amplified and thus the more distinct, endgame specific curve during the last part of the simulation in Figure 10b when the Oracle is used by our peers.

Figure 11a shows the achieved download speeds of the peers during this simulation as a boxplot containing the median, 50th- and 95th-quantile. The peers are grouped together by the network size of their ISP, the first two boxes being all peers of all networks. Each two boxes correspond to the same simulation setup, the first without any interaction of the Oracle, the second being the simulation where peers in our own network contacted the Oracle for assistance prior establishing connections with other peers. The improvements of the different networks are again clearly visible but only in the case of our own network and the Tier-1 network the 50th- and 95th-quantiles differ significant between the simulations. Thus even the fastest peers in bigger networks see an increased download speed. While the average download speed for the Tier-1 networks were indeed increased and the lower whisker stay roughly the same, the upper one actually decreases, which means that the fastest peers in small networks actually experience a decrease in download speed and the gain of the slower peer is at the expense of the faster ones. For peers in Tier-2 networks the distribution of achieved download speeds stays nearly identical in both simulations.

From a network operator’s perspective the gains when offering the service of an Oracle server are also quite good. The overall traffic flowing into our network is dropping by nearly 20% while the traffic flowing out of our network is decreased by around 4%. Most of the traffic savings were achieved during the later stages in the simulation as can be seen in Figure 11. The late savings in peering bandwidth with the help of the Oracle are due to the peering bandwidth being allocated for peer-2-peer traffic is saturated in the earlier stages of the simulation and the more peers inside our network become seeds the more the traffic is localized and thus stays inside of the network.

Most interesting for the network providers are the capabilities of the Oracle in respect to engineer the traffic flows. Figure 11b shows the average number of flows grouped by type (for distinction of the different flow types see chapter 7). The number of internal traffic flows is increased by roughly 70% while the number of incoming traffic flows is reduced by 40%. The number of outgoing traffic flows is decreased very slightly, only 5% less in comparison. Thus the Oracle allows a network provider the actively engineer the traffic flows such that the internal traffic flows increase and the incoming flows decrease. Only the impact on the outgoing traffic flows is quite small, for an explanation on that matter see chapter 7.
Figure 11: simulation: peering bottleneck, seed own AS
7.2 Case 2: Congested peering links with seed in Tier-3 network

In this experiment the position of the initial seed was changed in contrast to the one before to see how the Oracle can help when the seed is in a small network with poor connectivity. As a matter of fact, this also led to an increase in download time for all peers and such to an increase in time needed to be simulated. To clarify the differences, all changed parameters for this simulation are:

- The initial seed is placed in one of the Tier-3 networks and is a normal client having 16 MBit/s download and 1 MBit/s upload capacity.
- The simulated time was 20000 seconds, which was fairly big enough for everyone to finish the anticipated download.

Again the usage of the Oracle yields an improvement in achieved download speed for all peers, but the increase seen was roughly 4-5% regardless of the network a peer resides in (Figure 12a), thus the effect of the Oracle is not exclusively larger for peers in our own network.

In Figure 12a the average download speed over time for peers in our network can be seen. In the beginning you can see a steep rise in download speed which shortly after falls back to an even level for the rest of the simulation. This effect can be explained as follows: due to the peering bottleneck, the amount of content flowing from the seed into our network is limited. In the beginning the peering capacity is divided among fewer peers inside our network, thus the higher average download speed in the beginning. After more peers join the swarm, the rate at which new content is flowing into our network is slower than the available upload capacity of the peers. Thus the average download speed is mainly limited by the peering bottlenecks in the later stages of the download. In the case of the Oracle this is also true, but due to the localized peer matching the distribution within our network happens faster as new pieces are served on local connections thus avoiding the peering bottleneck. This effect has some similarities with a proxy server: few peer pull content from the seed into our network and then serve that content internally on the localized connections that were established with the Oracle's assistance. But with the
peering bottleneck becoming more concise the more peers join the swarm, the peers can not pull the content as fast as they could serv it to the local peers. Thus with more and more peers joining the swarm this “proxying” effect gets entirely neglected but still allows for an slightly higher average download speed nearly all the time throughout the simulation added to the magnified proxy effect in the beginning of the simulation. The distribution of achieved download speed can be seen in Figure 13a and has similar improvements for the different networksizes. The upper whiskers are slightly higher while the lower ones are increased quite a bit. Thus the increase in download speed is noticeable for all peers but slower ones see a higher increase than faster peers. The improvement in peering traffic is also visible but not as high as in the simulation before. The amount of traffic flowing into our network is lowered more than the amount of traffic flowing onto foreign networks. The average congestion of the peering links during the simulation can be seen in Figures 13c and 13d. Still the possibilities to engineer the traffic flows in the network operating the Oracle server are clearly visible in Figure 13b: while the number of internal traffic flows could be increased by only 17% due to the fact that the content has to be pulled from a foreign network into ours first, the number of incoming traffic flows was reduced by roughly 30%. In this scenario also the number of outgoing traffic flows was reduced by 10%, mainly due to the fact that if no content is available in our network it simply can not flow into foreign networks.
(a) overall download speeds of peers grouped by network size

(b) average number of unique traffic flows per type

(c) average congestion level for incoming peering links

(d) average congestion level for outgoing peering links

Figure 13: simulation: no peering bottleneck, seed small AS
7.3 Case 3: Uncongested peering links

In this chapter two experiments will be discussed at once as their results show nearly no difference in terms of significance. In both simulations all peering links between all networks do not pose a bottleneck and have a capacity high enough to allow all peers to use its complete upload capacity without creating congestion levels higher than 10% on any peering link. The only difference is the initial position of the seed, in one simulation the seed was in our own network, while in the other one the seed was located in a Tier-1 network. In the case where the seed initially is in our own network the average download speed is increased by roughly 7% for peers in our network and 5% for the others while it is increased by only 1% on average for all peers if the seed is in a Tier-3 network. This difference in increase is due to the fact that there is no proxying effect like it was when a peering bottleneck was present and such the localized connections do not yield any better performance because connections to foreign peers can utilizes their complete upload as well. Only when the seed is local the denser connections between our peers speed up the distribution among our own peers and thus yield a small increase. The average download speed over time plot in Figure 14b shows a compression of the original curve with slightly increased values. This again hints at the faster local distribution of pieces among peers in our network. This is the case for both simulations.

The number of unique flows as seen in Figure 15b again shows the traffic engineering capabilities of the Oracle: internal flows were increased by roughly 70% while the incoming traffic flows decreased by around 40% in both cases. The number of traffic flows going out of our network are also decreased in this setup by roughly 15%. While the explanation for the internal and incoming traffic flows is quite obvious because the Oracle alters exactly those connections, the drop in outgoing traffic flows is not. This is because every connection that is localized with the help of the Oracle service instead of going to a peer in a foreign network can not carry data out of our network and thus reduces the amount of traffic flows. The incoming traffic in both cases is reduced by approximately 18% and thus yields a quite good improvement. The outgoing traffic on the other hand is not lowered by such a huge margin but still yields 6% less traffic on average. The effect of the endgame is amplified when the Oracle is used, most
notably in the outgoing peering links, mainly because the Oracle allows to distribute the pieces faster among our peers and as a result the traffic increases if peers in foreign networks enter the endgame phase and request all left pieces from all available peers and because the outgoing traffic is not limited by the peering bandwidth. In the case of the seed being in the Tier-1 network, the endgame effect is also visible in the incoming traffic but is slightly flattened by the use of the Oracle due to the localized distribution of pieces and thus less requests of pieces from foreign peers. The plots for the incoming and outgoing congestion levels over time for both cases can be seen in Figure 16.
(a) average congestion level for incoming peering links when the seed is in a Tier-3 network
(b) average congestion level for outgoing peering links when the seed is in a Tier-3 network
(c) average congestion level for incoming peering links when the seed is in our own network
(d) average congestion level for outgoing peering links when the seed is in our own network

Figure 16: simulation: no peering bottleneck, seed small AS
Figure 17: a

7.4 Case 4: Overprovisioned seed

The experiments discussed in this section were done with an overprovisioned seed to see the effects the oracle can have in a scenario where maybe a commercial distributor wants to add additional well provisioned data sources to the swarm to speed up content distribution among his customers. When the overprovisioned seed is placed is positioned in a Tier-3 network it has 10 Mbit/s upload capacity, while in the case where it is positioned in our network it has 100 Mbit/s upload capacity. This is because we are in this case able to control the resources allocated to the data source and have a better estimate on the congestion levels inside of our network opposed to a foreign network. If the seed is in a Tier-3 network, the simulations were done with and without the presence of a peering bottleneck, if the seed is in our network peering bottlenecks were also present.

The improvements in average download speed (see Figure 17) were 36-40% on average for peers within our network (around 15% for peers in other networks) without peering bottlenecks being present and roughly 15% for all peers regardless of which network they were in if peering bottlenecks were present, thus one limiting factor in a case with an over provisioned seed is the peering bottleneck. The average achieved download speed distribution is changed only very little if the over provisioned seed is placed in our network: nearly identical for peers in foreign networks with slightly increased whiskers but well increased lower and upper whiskers for peers inside our network. Thus the speed increase is available for slow and fast peers alike. This is also true if the seed is in a foreign network and no peering bottlenecks are present. If peering bottlenecks are present this picture changes: the highest increase in average download speed is experienced by slower peers while the faster ones stay nearly identical. This can be attributed to the better local pieces distribution, which helps peers with fewer pieces more as they can get them from local peers, while the fast peers with more pieces have to grab the remaining ones from the seed in a foreign network and thus are limited by the peering bottlenecks. This can also be seen in the average download speed over time (Figures 18a and 18b) as the case with the seed in a foreign network and peering bottlenecks present is the only one differing: while the other two are basically compressed versions of the original curve with an increased throughout the complete simulation, this simulation again
shows a proxy-like effect, where in the beginning the average download is much higher due to good local availability of pieces which decreases the more local peers join the swarm.

The traffic levels on the peering links are again quite similar in the case with the seed in our network and the case were the seed is in a foreign network but no peering bottlenecks are present (Figures 19a and 19b). In the beginning the incoming traffic levels rise and are limited by the peering bottlenecks but congestion levels are reduced constantly throughout the simulation as content becomes more available on local peers. The outgoing traffic levels are bounded by the peering links capacities until almost every peers has finished the download. Traffic savings are most notably on the incoming traffic and are around 14% less incoming traffic, while the outgoing is slightly reduced in the case where the seed is in our network (5% less outgoing traffic) while it stays roughly the same in the other cases.

In the case with peering bottlenecks present (Figures 19c and 19d) the incoming traffic is increases in the beginning when the Oracle is used but drops early towards the end of the simulation because the pieces are available locally and overall saves traffic in the neighborhood of 5%. The outgoing traffic is increased by roughly 4% in this scenario mainly because due to the Oracle the pieces are available in our network faster and thus are also requested faster and by more peers outside of our network, but due to the faster distribution of pieces toward foreign networks drops earlier as peers finish their downloads earlier.

The number of different flow types (see Figures 18c and 18d) are quite similar for both cases where the seed is in a foreign network, the internal flows were increased by 75-80% and the incoming flows reduced by roughly 30%, while the incoming flows slightly increased by 10% due to the fact that with the help of the Oracle the pieces can be pulled faster inside into our network due to the recommended peers having higher bandwidth. In the case where the seed is in our own network the difference is that the increase in localized traffic flows is lower, only 40% increase in localized traffic carrying connections. But we were able to reduce the number of traffic carrying flows out of our network by around 15% due to more connections being localized and thus peers establishing and accepting fewer connections from peers in foreign networks.
(a) average download speed over time per client in our own network with an overprovisioned seed in our network, peering bottlenecks present red: no oracle, green: oracle

(b) average download speed over time per client in our own network with an overprovisioned seed in a Tier-3 network, peering bottlenecks present red: no oracle, green: oracle

(c) average number of unique traffic flows per type, overprovisioned seed in our network, peering bottlenecks present

(d) average number of unique traffic flows per type, overprovisioned seed in a Tier-3 network, peering bottlenecks present

Figure 18: simulation: overprovisioned seeds
Figure 19: Simulation: Overprovisioned Seeds
This section will describe some experiments with multiple torrents per peers downloaded in parallel. This time we repeated this experiment in three different settings regarding the Oracle service:

1. No Oracle is used
2. The Oracle is used for both downloads
3. The Oracle is only used for only one download

While setup 1 serves as a base setup to compare the Oracle against and 2 is the setup where every established connection is done with the help of the Oracle, setup number 3 was done to see how the Oracle influences downloads which do not make use of the service offered by the Oracle. This is a quite important scenario during the transition of network providers offering the Oracle service to its customers, as not every application will come with Oracle support and not every end user might actually want to use the service regardless of the benefits such a service provides to them.

The settings for the network and the BitTorrent configuration were the same for all conducted simulations:

- Peering bottlenecks are present between all participating networks
- Each peer participates in both downloads
- Both files downloaded were 20 MByte in size and the piece size was set to 128 KByte
- Both seeds are in our network and are normaly peers having 16 MBit/s download and 1 MBit/s upload capacity
When the Oracle is used for both torrents the peers download the improvements on both downloads are quite similar. Figure 20a shows the achieved average download speed of one of the two downloaded torrents. If the assistance of the Oracle was requested for both torrents the increase seen by peers using the Oracle was 22% for the first torrent and 12% for the second one. Peers not using the Oracle also gained between 2% and 15% on average. Thus the Oracle is able to improve multiple parallel downloads by advertising better peer connections to its customers.

Even more interesting is the case where only one of the two torrent downloads made use of the Oracle’s service: The peers using the Oracle for the first torrent saw an increase of roughly 40% on average, while the second torrent which did not use the Oracle saw a small decrease of 3% on average. Peers not in our network and thus not using the Oracle at all saw an increase in average download speed of between 11% and 19% for the first torrent and also a slight decrease of 3% in download speed on average. This shows that the influence of the Oracle on downloads not using the Oracle is minor and thus does not interfere with users that do not want to make use of the service offered by the Oracle.

The traffic savings in the case where only one torrent used the Oracle were roughly 8% on incoming peering traffic and 3% on the outgoing, in the case where both torrents used the Oracle the savings increased to 11% and 4% respectively. While the amount of saved traffic on the peering links seems quite low, it is important to remember that both seeds were in our network and that the peers requesting data from peers in our network could not use the Oracle and that the absolute amount of incoming traffic was already quite low in the original BitTorrent simulation. More important are the figures showing the average number of traffic carrying flows which can be seen in Figure 21b. The improvement in the different types of flow are nearly linear in the sense of how many torrents made use of the Oracle: with only 1 torrent using the Oracle the number of internal flows increased by 16% while it increased by 32% when both torrents contacted the Oracle. The same applies for the number of incoming and outgoing flows: -17% and -7% if one torrent made use of the Oracle and -30% and -13% if both did. Thus the traffic engineering capabilities of the Oracle do work perfectly if a peers participates in multiple parallel downloads even if only some of them contact the Oracle for assistance.
Figure 21: simulation: no peering bottleneck, seed small AS
8 Conclusion and Future Work

This thesis has shown that the proposed Oracle concept [2] does have quite an impact on the BitTorrent overlay network and does increase the performance of the end-users in terms of faster downloads as well as the ability of the network operator to actively engineer traffic flows within its network. Although the amount of peering traffic saved by the use of the Oracle service is substantial it is still quite low in comparison to the savings that the Oracle achieved in the Gnutella overlay network [25]. This is mostly attributed to one fundamental design in the BitTorrent protocol: each peer that wants to request missing pieces of a download has to upload to the peers it is connected to in order to be able to actually request and download desired pieces.

Another major contribution of this thesis is the impact of different parameters on the results the Oracle can provide. The parameters analyzed include (but are not limited to): the position of the initial seed, presence of peering bottlenecks, overprovisioned seeds and the number of parallel downloads. With better understanding of the impact of different parameters on the results it is possible to tailor specific and optimized ranking functions to meet the desired goals of the network operator providing the Oracle service while delivering increased performance to the end-users at the same time. In all cases the improvement for both, the end-user and the network operator, was clearly visible. For the end-user the most critical performance is the achieved download speed and thus the time it takes to complete a download. In no case did the usage of the Oracle degrade the download speed of the peers seeking the Oracle’s assistance and in all cases the performance increase was either very high (around 15%-20%) or moderate (between 3% and 5%). For the network operator the ability to engineer the traffic carrying flows between two peers was always tremendous although BitTorrent is not the best protocol for such engineering because of some fundamental design properties that requires always both peers to participate in data exchange. Still the numbers are quite promising as the number of network internal data flows can always be increased from 30% up to 70%, while the number of outgoing data flows are decreased by 25%-40%. This thesis has provided an increased understanding of the Oracle and the possible improvements it can achieve and thus provides a platform that future Peer-to-Peer protocol designs can take advantage of to further improve the performance for all involved parties.

In all experiments the ranking function of the Oracle uses the capacity of the links when gathering the information about bandwidth on the path between two peers. Most of the experiments were re-run with the Oracle using the available bandwidth on the path between two peers instead of the link capacities, thus using a slightly different ranking function. The results of this modified ranking function still yields an improvement in performance for all involved parties but the gains were lower. This is due to the fact that the available bandwidth of a link changes quite fast over time while BitTorrent does not utilize a new peer connection directly after establishing it but when the peer becomes unchoked, which might happen very soon but also quite late in time. Another important fact is that getting the available bandwidth information (or rather the congestions level) of a link into the Oracle is quite hard to establish and without a significant additional gain in performance this is considered too much overhead.

Although this thesis was able to answer most of the questions regarding the influence of different parameters on the Oracle and how this information can be used to tailor an application specific ranking functions there are still a few open questions. One aspect this thesis did not take into account was the presence of node-churn. Because of the flash-crowd type scenario in combination with a relatively small simulation time (roughly 3 hours of real time), we considered that node churn would take place mostly in the end of the simulation and thus would not change the results drastically. It still needs to be determined how node churn would influence the Oracle’s performance in longer living torrents or much sparser swarms. Sparse swarms would be another area that would be worth investigating further as well as the user’s behaviour of joining and leaving the swarm. Both factors were quite static in our simulations, the swarmsize was fixed and the joining behaviour was modelled with a uniform distribution. The uniform distribution was chosen because of the flash crowd like scenario, during 300 seconds all 720 peers joined the swarm, thus the impact of different behaviour distributions would be quite small. Another interesting aspect is the number and distribution of multiple torrents among peers. This would be another interesting area of research. In most of our scenarios every peer was participating in the same torrent swarm thus
the oracle was used to optimize a single overlay network. Only one experiment exploring the impact of the Oracle on multiple parallel downloads was conducted and yielded quite interesting results though time was the limiting factor exploring this area any further. Last but not least the settings of a torrent download, such as the overall file size, the piece size, multiple trackers or even trackerless torrents would pose additional parameters worth investigating.
References


