The Need for Distributed Tracking

Raynor Vliegendhart

Delft University of Technology

The Need for Distributed Tracking

Research Assignment Computer Science

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Raynor Vliegendhart

6th May 2009

Preface

This document is the report of my research assignment, the preparation for the Master thesis assignment in the Parallel Distributed Systems Group of the Delft University of Technology. The report reviews the possible distributed solutions to the swarm discovery problem for the BitTorrent P2P network, in order to replace the current central tracker solution.

I would like to thank my supervisor dr. ir. J. A. Pouwelse for his advice and guidance. I would also like to thank Victor Grishchenko for information concerning the PEX experiments he performed.

Raynor Vliegendhart

Delft, The Netherlands
6th May 2009

Contents

Preface v

1 Introduction 1

2 Problem Definition 5
- 2.1 Swarm discovery . . . 5
- 2.2 Deployed solutions . . . 6
- 2.3 Problem aspects . . . 7
  - 2.3.1 Connectability . . . 7
  - 2.3.2 Churn . . . 7
  - 2.3.3 Security . . . 8
  - 2.3.4 Bootstrapping . . . 8

3 Workload Analysis on Public BitTorrent Trackers 9
- 3.1 Measurement methodology . . . 9
  - 3.1.1 Terminology . . . 9
  - 3.1.2 Data acquisition . . . 10
  - 3.1.3 Computing the number of peers per tracker metric . . . 10
- 3.2 Results and discussion . . . 11

4 Distributed Hash Tables 13
- 4.1 An introduction to DHTs . . . 13
- 4.2 Discovering the swarm with a DHT . . . 14
- 4.3 Problems encountered in DHTs . . . 14
  - 4.3.1 Connectability . . . 15
  - 4.3.2 Churn . . . 15
  - 4.3.3 Security . . . 17
  - 4.3.4 Bootstrapping . . . 19
  - 4.3.5 Load balancing and performance . . . 20

5 Gossip Protocols and Graph Walks 23
- 5.1 Introduction to gossip protocols and graph walks . . . 23
  - 5.1.1 Gossip protocols . . . 23
  - 5.1.2 Graph walks . . . 24

5.2 Discovering the swarm through gossips and walks 26
5.2.1 Peer Exchange (PEX) 26
5.2.2 Random walks 27
5.3 Problems encountered with gossips and walks 29
5.3.1 Connectability 29
5.3.2 Churn 30
5.3.3 Security 31
5.3.4 Bootstrapping 31

6 Proposed Distributed Tracking Algorithm 33
6.1 Environment 33
6.2 2-Hop TorrentSmell 33

7 Conclusions 37
7.1 Conclusions 37
7.2 Further research 38

Chapter 1

Introduction

Peer-to-Peer (P2P) applications are quite popular and currently responsible for a large part of Internet traffic (see Figure 1.1). The most notable use of P2P applications is file sharing, enabling easy, fast and widespread distribution of a wealth of content.

The challenge in developing P2P applications is to make them scalable, secure, fault-tolerant, and self-organising. Fault tolerancy and self-organisation of a a P2P system can be maximized when it is fully decentralised. The past, however, has shown that designers of popular P2P systems often sacrifice full decentralism by including central components in their design. Regardless of the underlying reasons, these choices for centralism have lead to the downfall of popular services, such as Napster [19] and the popular BitTorrent tracker Supernova [43], and are good examples of the exploitation of *single point of failures*.

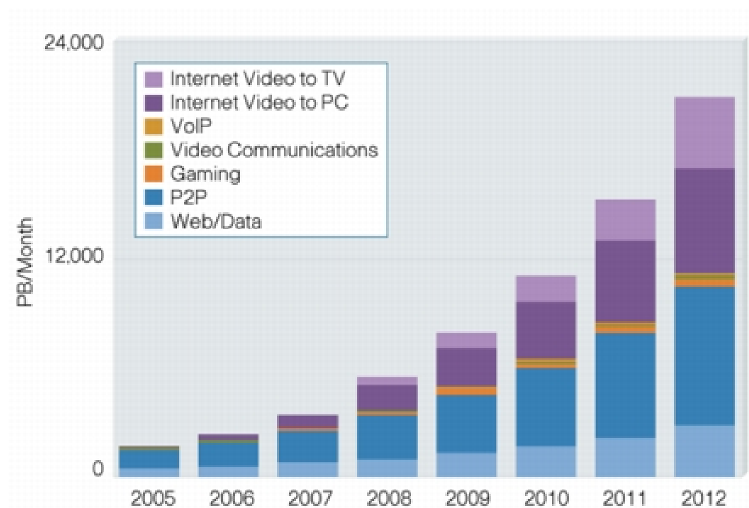


Figure 1.1: Global consumer Internet traffic forecast. Source: Cisco, 2008 [8].

The central component of the BitTorrent protocol is called the tracker and is responsible for tracking peers in the system. Peers use trackers to discover other interesting peers. The central nature of the tracker motivates the research presented in this report. In order to let BitTorrent evolve into a fault-tolerant self-organising P2P system with no bounds to scalability, we must replace the protocol's final obstacle to full decentralisation.

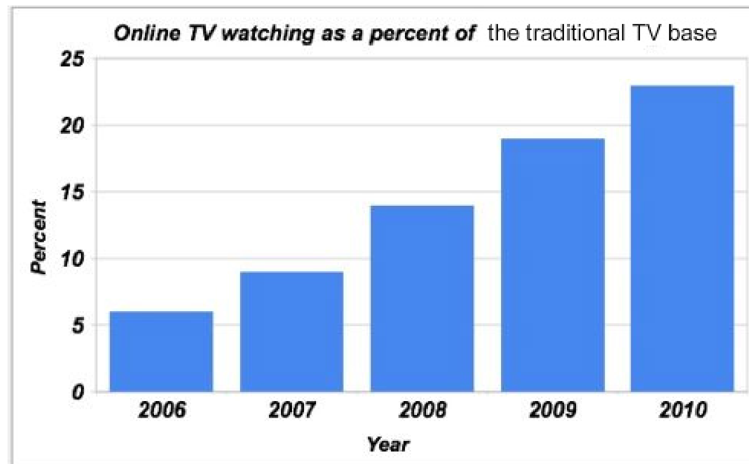


Figure 1.2: Percentage of traditional TV base in the US that watch TV online. Source: Convergence Consulting, 2008 [24].

The unbounded scalability we aim for is that of television level. There are more than 1.4 billion televisions in use worldwide [23] and today more than one billion Internet users exist [13]. An unbounded scalable P2P network should thus be able to serve at least these many users. It is not unreasable to set such high aims. Currently, popular broadcasted shows already reach millions of unofficial downloads [37, 39] and the current trends show that traditional television is moving online (Figure 1.1, Figure 1.2). In Australia, people are even already spending more time online than behind their television set (Figure 1.3). Obviously, this move to the online world has significant consequences for existing business models.

The goal of this report is to explore the design space of a distributed alternative for BitTorrent's tracker mechanism in order to serve hundreds of millions of peers without the need of any server. Zero-server P2P is the key aspect of the 4th Generation of P2P [41]. Our research will serve as a basis for future implementation and incremental deployment in Tribler, a social BitTorrent client developed by Delft University of Technology and Vrije Universiteit [40]. Hence, we will assume a Tribler-like environment.

The remainder of the report is organised as follows. In Chapter 2, we give a definition of the problem to be solved that central trackers currently solve, namely swarm discovery. In Chapter 3, we analyse the current situation in the BitTorrent world

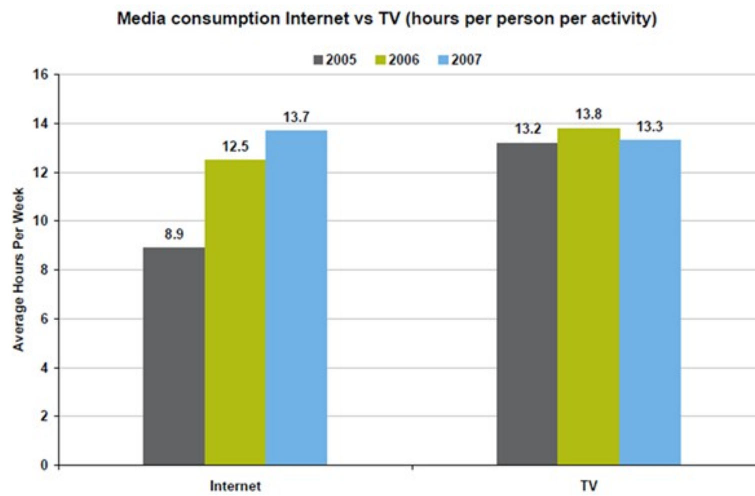


Figure 1.3: In Australia, usage of Internet has surpassed traditional television. Source: Nielsen Online, 2008 [20].

and present results of a large scale public tracker measurement. In Chapter 4, we explore Distributed Hash Tables and study how they are used to solve the swarm discovery. We do the same for gossip protocols and graph walks in Chapter 5. Using the knowledge gained in the previous two chapters, we will propose our own distributed solution to the problem in Chapter 6. Finally, we conclude our report with a summary of our findings and discuss future work.

Chapter 2

Problem Definition

In this chapter we define the research problem subject to research: swarm discovery in a secure and scalable way. Swarm discovery is the problem of finding peers that are downloading the same file, which we describe in Section 2.1. We give a short overview of the current deployed solutions to this problem in Section 2.2. Solving the problem in a distributed setting is complicated by dynamic aspects, which we cover in Section 2.3.

2.1 Swarm discovery

In P2P networks, resources and information about these resources reside at peers, unlike traditional client-server networks in which they reside at a central computer. Since there is no central place to look up, peers are burdened with the task of locating both peers and resources themselves. This task is complicated by the dynamic nature of P2P networks, in which peers can join and leave at any time.

In BitTorrent, peers are interested in locating other peers that are downloading the same content, allowing them to barter file pieces. These peers are said to be in the same *download swarm*. In previous work done by Roozenburg [30], *swarm discovery* is defined to be the problem of finding all peers in a specific swarm.

The information needed to communicate with another peer is that peer's public

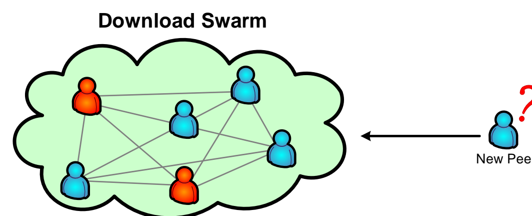


Figure 2.1: Swarm discovery is the problem of finding nearly all peers in a specific swarm. In the depicted situation, the new peer must somehow acquire contact information of the peers in the download swarm.

IP address and listening port. A list of network addresses of peers in a certain swarm is conveniently called a *peer list*. A peer list can be obtained in several ways, like from example from an external source or from peers already in the swarm. In the situation depicted in Figure 2.1, the new peer does not know anything about the swarm yet and must rely on some external source.

2.2 Deployed solutions

Currently in BitTorrent, three solutions are deployed to solve the swarm discovery problem:

- One or multiple central trackers
- Distributed Hash Table (DHT)
- Peer Exchange (PEX)

Central trackers have been used since the first version of BitTorrent. A swarm can be tracked by one or multiple trackers. Peers know of these trackers via a swarm's metadata stored in a `.torrent` file and they regularly contact a tracker to request a random sample of the tracker's peer list and to announce their own presence in the swarm [9]. The process of joining the swarm is shown in Figure 2.2.

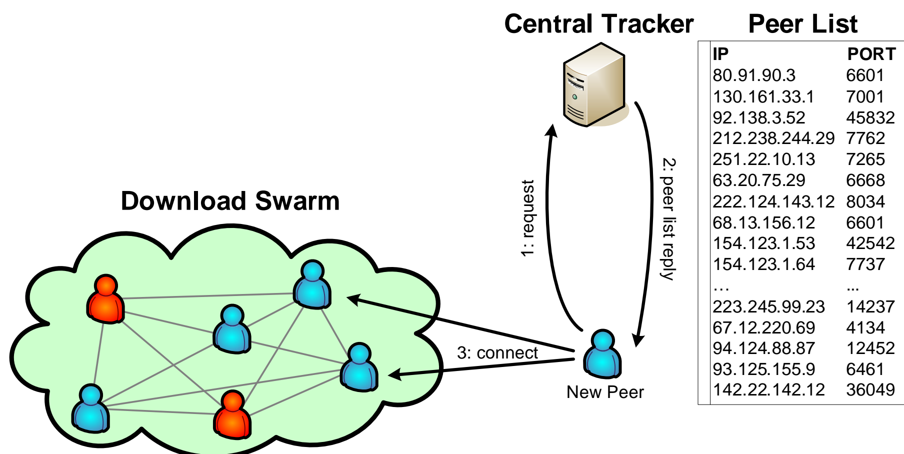


Figure 2.2: The swarm discovery problem can be solved using a central server called a tracker. Here, a new peer uses the tracker to find peers to connect to.

Instead of using a central server to store a swarm's peer list, each swarm's peer list can also be stored in a Distributed Hash Table. There are two BitTorrent DHTs in use today [3, 6], which we discuss in Chapter 4.

Alternatively, a peer can also exchange peers (PEX) with its connected neighbours in order to discover a larger portion of the swarm. We come back to this topic in Chapter 5.

2.3 Problem aspects

In this section we discuss the dynamic aspects of the problem. These aspects are complicating the swarm discovery problem and *must* be dealt with if we want to solve it effectively and securely.

2.3.1 Connectability

Connectability is one of the important aspects of the swarm discovery problem. In a perfect system all peers in the peer list are connectable. However, efficiency is lost when only a minority is connectable. Trackers can validate the connectability by attempting to connect to announcing peers. Peers which are not connectable are not included in the tracker's peer list.

An example of efficiency loss can be found in *BuddyCast*. In an early version of this epidemic gossip protocol, nearly 60% of the messages were unreliable, containing peer addresses of which 81-100% were either offline or unconnectable [27]. As a result, much time and bandwidth were wasted on futile connection attempts to an unconnectable peer. Later versions implemented a mechanism allowing a peer to check its own connectability through a *dial back message*. Information acquired from such a test allowed unconnectable peers to inform others not to propagate their IP addresses, resulting in reliable messages.

2.3.2 Churn

The dynamic nature of P2P networks is that peers can join and leave at any time. The rate at which this happens, i.e. the number of joins and leaves per time unit, is called *churn*. Churn has a great effect on the operation of a P2P network. In structured networks, churn complicates maintaining the network topology. In P2P networks in general, it causes information to become outdated: peers said to be online may actually have left the system in the meantime.

In the swarm discovery problem, this means that it is important to know how old your information is. Peers in a received peer list may no longer be online and trying to connect to them may not work.

So when does peer list data become stale? We may find our answer in the extensive research on churn done by Stutzbach and Rejaie [33]. Based on measurements in three BitTorrent swarms and in two other P2P networks, the authors conclude that a large portion of active peers is actually highly stable. In two Linux ISO swarms, the probability of a peer being up for more than 5 hours was 60%. The remainder, however, consists of short-lived peers that join and leave the system at such a high rate that they constitutate a relatively large portion of sessions. That is, they regularly return for unfinished business. The authors also note that a peer's uptime is a good indicator of its remaining uptime, but exhibits high variance.

2.3.3 Security

Security is an aspect that is often overlooked when designing distributed solutions, yet it is an important matter. Taking security issues lightly we may end up in a situation where our P2P network is ineffective, or even worse, exploitable.

One of the possible attacks on a P2P network is to pollute or poison its indexes, as described by Liang et al. in [18]. Projecting this type of attack onto the swarm discovery problem, it basically means that an attacker is spreading bogus peer lists, trying to make it harder for other peers to find a valid entry in their lists. To defend against such attacks, the authors propose a rating system, e.g. to rate the source of information.

A much more severe issue is exploiting the vast amount of resources available in a P2P network to perform a *Distributed Denial of Service* (DDoS) attack. Since some P2P networks consist of millions of users,[1] harnessing all this power would mean havoc to the victim. To avoid such attacks, an attacker must be limited in its ability to spread false information. One possibility could be having peers validating information the moment they would receive it, but such an approach is also vulnerable to attacks [34].

2.3.4 Bootstrapping

Bootstrapping is a special startup process one performs in order to become self-sustaining. For example, a personal computer *boots* by loading a special program from the first sector of a disk. Afterwards, the computer will be able to operate normally.

In the case of swarm discovery, a peer in the bootstrapping phase has to find a small number of peers that are in the swarm or peers that know about them. These initial peers can then be used to operate normally by executing the swarm discovery algorithm. If these initial peers cannot be found, further swarm discovery becomes impossible.

Note that when a centralized solution like a central tracker is used, no bootstrapping is necessary. The address of the tracker is always known and thus both initial and subsequent peers can always be found using the tracker. In a decentralized setting, however, a bootstrap step is always required.

[1]For example, Azureus' DHT consists of more than 1 million nodes [14].

Chapter 3

Workload Analysis on Public BitTorrent Trackers

As mentioned in the previous chapter, the swarm discovery problem in BitTorrent is currently mainly solved by central trackers. These trackers are the bottleneck preventing BitTorrent to become truly scalable and fault-tolerant. However, we must also investigate how pressing this matter is. To find out, we performed a large measurement on the activity of public BitTorrent trackers.

In Section 3.1, we describe how we measured the popularity of numerous trackers and their relative importance to the download swarms by computing the number of peers each tracker tracks. In Section 3.2 we present and discuss the results of this measurement.

3.1 Measurement methodology

For each tracker encountered during the measurement, we computed the number of peers it tracks. The definition of *peer* and *tracker*, among others, used in this computation are given in Section 3.1.1. How we acquired the data on which we performed the computation is explained in Section 3.1.2. The computation of a tracker's size itself is described in Section 3.1.3.

3.1.1 Terminology

Swarm A group of peers downloading and sharing the same content.

Peer A peer is a BitTorrent client that is participating in a specific swarm. A client that is connected to n swarms counts as n distinct peers.

Torrent A file containing information on a swarm, such as the filename of the content and the primary and, if any, backup tracker addresses.

Tracker A server responsible for keeping track which peers are in a swarm. Multiple tracker servers operating under the same name or using the same domain name are counted as a single tracker.

Swarm type Swarms can be tracked by either a single tracker or by multiple trackers, and we call thus these type of swarms *Single Tracker* and *Multi-Tracker*, respectively. From the tracker's point of view, a swarm of the latter type can be distinguised to be either *Primary* or *Foreign*, based on whether the tracker is listed as the primary tracker in the torrent file.[1] When we refer to a *peer's swarm type*, the swarm type of the swarm the peer resides in is meant.

3.1.2 Data acquisition

We collected thousands of torrent files over a time span of fourteen months, using Tribler's so called *TorrentCollecting* feature. Tribler clients contact each other regularly and exchange newly found torrent files. For each known swarm, Tribler regularly sends requests to the associated trackers to query the swarm size, i.e. the number of peers in that swarm. This information is stored in a database called the *MegaCache*.

3.1.3 Computing the number of peers per tracker metric

To compute the number of peers each tracker tracks, we built a bipartite graph of trackers and swarms as follows:

1. For each swarm in the MegaCache:
   (a) Add the swarm to the graph.
   (b) Extract the hostnames from the tracker addresses.
   (c) For each hostname:
      i. Perform a DNS lookup and determine whether the tracker is already in the graph. If not, add it.
      ii. Add an edge between the swarm and the tracker, using the swarm size as the edge's weight.
   (d) Determine the swarm type.
2. Merge related tracker nodes in the graph into a single node, using a manually crafted mapping.[2]
3. Remove swarms with incorrectly reported swarm sizes, e.g. swarms with 0 peers or less.
4. Remove any trackers that no longer track any swarms.

We then ranked each tracker according to the total number of peers they tracked.

[1] We use the `announce` field in the torrent file to determine which tracker is primarily responsible for tracking peers in the swarm.

[2] Using DNS to group hostnames does not work in all cases. For example, Sumotracker's two domains `sumotracker.com` and `sumotracker.org` were merged manually.

3.2 Results and discussion

Figure 3.1 shows the results of our measurement which covered 283,032 torrents. From this chart, it is clearly visible that The Pirate Bay (TPB) [35] is the most prominent central tracker in use.

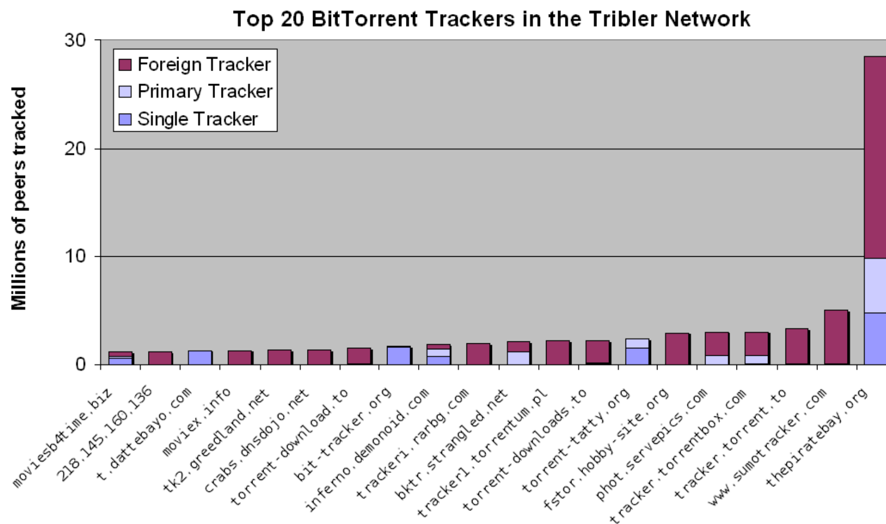


Figure 3.1: The top 20 public BitTorrent trackers encountered in the Tribler network ranked by the number of peers they track. Peers are distinguished by swarm type. The dataset consists of 283,032 download swarms with a total of 52,634,778 peers, collected over a time span of 14 months.

This has consequences for the landscape of public BitTorrent trackers. The Pirate Bay (TPB) benefits from having the largest market share and continues to attract users. Is there any reason not to use the largest and well known public tracker? Most users seem to answer this question with “no”. Their trackers already generate traffic of 600 Mbit/s and their costly hardware (approximatedly $120,000) needs to be replaced yearly to cope with growth [38]. These figures also illustrate that scaling a central tracker architecture is quite costly. TPB only seems to stay afloat through income from advertising income [29].

The existence of a single, large and dominating public tracker does give rise to concern. If the largest tracker fails, what will happen? In the general case for single tracker swarms, when a tracker fails, it will influence the number of downloaders in the swarms it was tracking, since evidently an important source for finding peers disappears [26]. However, when a large tracker is also responsible for carrying the load of tracking peers in multi-tracker swarms and fails, this load will carry over to other, smaller trackers. From Figure 3.1, we can see that the multi-tracker load for TPB is fairly large. If TPB fails, this transfer of load would mean a sudden increase in load on other trackers, especially when a swarm was tracked by only

a few trackers. As a result, response times on these trackers tend to be longer, or worse, these trackers may even go down.

Clearly, solely relying on a central tracker architecture seems to be a bad idea. It does not scale and in the current public tracker landscape it is not resilient to failure. To cope with these problems, BitTorrent developers have developed a form of distributed tracking, which we will examine in the next chapter.

Chapter 4

Distributed Hash Tables

In this chapter we analyse the use of Distributed Hash Tables (DHTs) to solve the swarm discovery problem. In Section 4.1, we explain how DHTs work using Chord as an example. In Section 4.2, we briefly describe how DHTs are currently used to do swarm discovery. Problems encountered in DHTs are discussed in Section 4.3.

4.1 An introduction to DHTs

A Distributed Hash Table (DHT) is a structured distributed system that offers a lookup service similar to a non-distributed hash table. An ordinary hash table stores (*key*, *value*) pairs. In a DHT system, different nodes are responsible for different key ranges. This responsibility can be distributed in several ways.

In Chord [32], the full key space consists of 2^m identifiers. File keys and nodes are mapped onto these identifiers in the key space by a consistent hash function (SHA-1). For a node identifier the node's IP address is hashed. The node identifers are ordered on a so called *identifier circle* (modulo 2^m), or *Chord ring*. Keys are assigned to their so called *successor nodes* as follows. The successor node of key k, denoted $successor(k)$, is the first node whose identifier equals or follows k. An example Chord ring is shown in Figure 4.1.

Simple but slow lookups can be implemented by having each node maintain a pointer to its successor. Upon a lookup request, a node checks whether it is the successor of the key being looked up. If this is the case, a reply is sent back with the node's identifier. Otherwise, the request is forwarded to the node's successor.

A scalable key lookup can be implemented by having each node n store a *finger table*. The ith entry in this table contains the successor of key $n + 2^{i-1}$. This table can thus be used to make large jumps in the Chord ring, resulting in an $O(\log N)$ message complexity.

Joining a Chord ring is achieved by computing one own's node identifier n and requesting a node that is already part of the ring to find the successor of n. The result of this query is then one own's successor. The predecessor is set to nil. All predecessor and successor pointers in the ring will eventually point to the correct

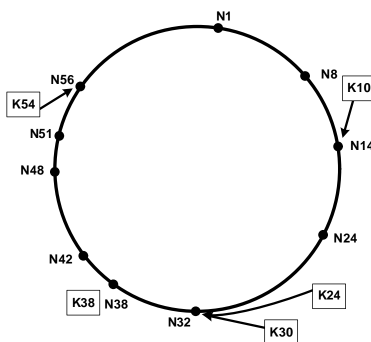


Figure 4.1: A Chord ring consisting of ten nodes storing five keys [32].

nodes by a stabilization process, which involves regularly sending notification messages to one own's successor and by periodically checking one own's predecessor. The finger tables are fixed by regularly performing a lookup for each table entry.

Other DHTs differ in e.g. how the keys in the key space are assigned to nodes and how messages are routed. In Kademlia, a DHT on which the currently deployed BitTorrent DHT implementations are based, an XOR metric is used for the distance between points, rather than the clockwise ring distance in Chord. Additionally, Kademlia nodes can choose their own random node identifier [21].

4.2 Discovering the swarm with a DHT

Currently, two BitTorrent DHTs are in use, namely the Mainline DHT and the Azureus DHT [11]. To avoid confusion, we will use "node" to mean a client/server implementing the DHT extension and "peer" to mean a BitTorrent client in a swarm. A node can be a peer in multiple swarms and can be responsible for tracking multiple swarms, but not necessarily swarms in which it is in. It is responsible for those swarms whose *infohash* is closest to its own node identifier.

When a peer wants to find other peers of a certain swarm, assuming it has already joined the DHT, it looks up the node that is responsible for tracking that swarm and then query it for a peer list. Afterwards it also announces its own presence to that node in order to update that node's peer list (Figure 4.2).

4.3 Problems encountered in DHTs

When using DHTs for swarm discovery, several problems need to be tackled. In addition to the aspects covered in Section 2.3, we will also discuss the problem of load balancing and performance in this section.

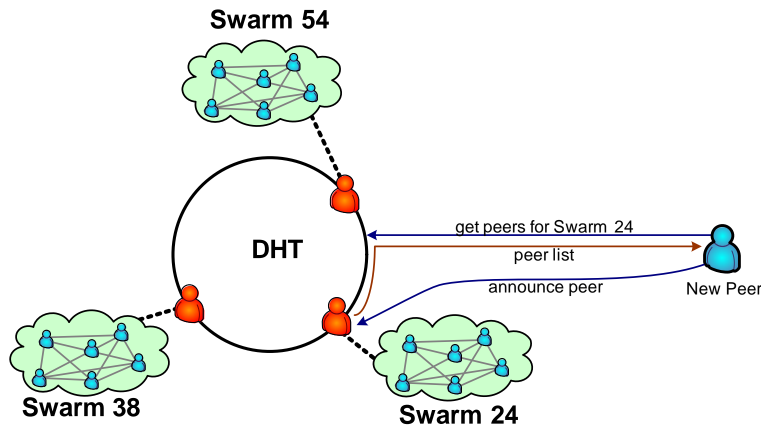


Figure 4.2: Finding peers in a swarm with a certain infohash involves looking up the DHT node that is responsible for tracking that swarm and querying that node for a peer list. After receiving a response, one's presence is announced so that the responsible tracking node can update its peer list.

4.3.1 Connectability

In the two deployed BitTorrent DHTs, nodes seem not to take into account whether they are connectable. A study by Crosby and Wallach [11] suggests that 10-15% of BitTorrent users have significant connectivity problems, likely resulting from firewalls or NATs. They found that certain hosts suffer from one-way connectivity. These hosts were contacting the authors' host multiple times, yet the authors were were unable to reach them, even after multiple attempts. Additionally, the number of times a node has been seen seems to say little about whether it reachable. Figure 4.3 shows there is virtually no correlation between the number of times we have seen a node (k-seen) and the number of successive failed attempts (j-unreachable).

4.3.2 Churn

Swarm churn causes peer lists to become stale, but this problem can be mitigated by having peers announce them regularly.[1] In structured P2P networks like a DHT, however, churn has a much greater impact. The arrival and departure of nodes weakens the structure of the network, and thus must be repaired.

There are two ways to recover from churn: reactive recovery and periodic recovery. The latter approach is used by Chord and the corresponding recovery process takes place independently of a node detecting changes in its routing table. With the former approach on the other hand, nodes react immediately to changes and inform their neigbours to fix the network topology. While the reactive recovery approach might seem preferable at first sight, it actually runs the risk of creating a positive feedback cycle as illustrated by the following example:

[1]In Kademlia, nodes are required to republish keys by design [21].

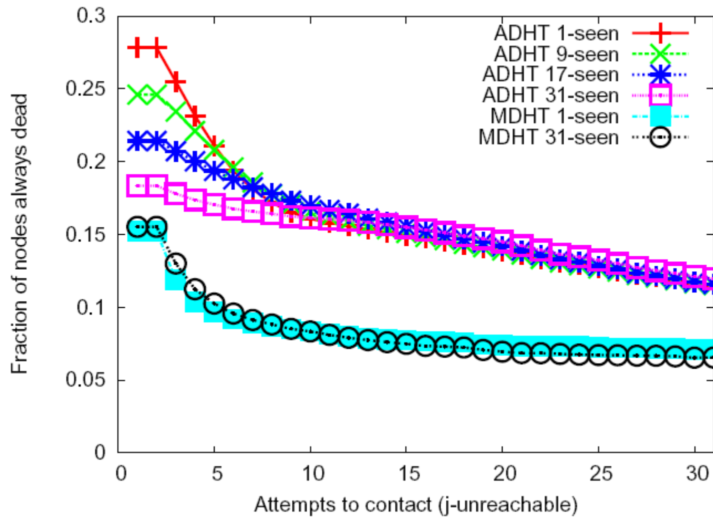


Figure 4.3: Unreachability ratio as a function of j-unreachable and k-seen for the Mainline DHT (MDHT) and the Azureus DHT (ADHT) [11].

> "Consider a node whose access link to the network is sufficiently congested that timeouts cause it to believe that one of its neighbors has failed. If the node is recovering reactively, recovery operations begin, and the node will add even more packets to its already congested network link. This added congestion will increase the likelihood that the node will mistakenly conclude that other neighbors have failed. If this process continues, the node will eventually cause congestion collapse on its access link." [28]

The authors of [28] measured the differences in bandwidth usage and latency between the two approaches in a Bamboo DHT under two different 20-minute churn periods, separated by a 5 minutes churnless period. The first churn period had a median of 47 minutes session times, while the second churn period had a median of 23 minutes. The results in Figure 4.4 clearly show that while reactive recovery is very efficient without churn, periodic recovery is to be preferred at reasonable churn rates.

How messages are routed in the DHT also influences the capabilities to cope with churn. Messages can be either routed recursively or iteratively. Under a recursive routing scheme, the queried node is responsible for forwarding the messages to the next node, while under an iterative scheme the querying node is responsible for sending all messages (Figure 4.5). In the two BitTorrent DHTs, which both use iterative routing, 20% of the nodes contacted in a single lookup are no longer alive on 95% of the lookups. Overall, about 40% of the encountered nodes are dead. These dead nodes cause an increase in lookup latency since each dead node incurs timeout overhead. The authors of [11] recommend switching to a recursive routing

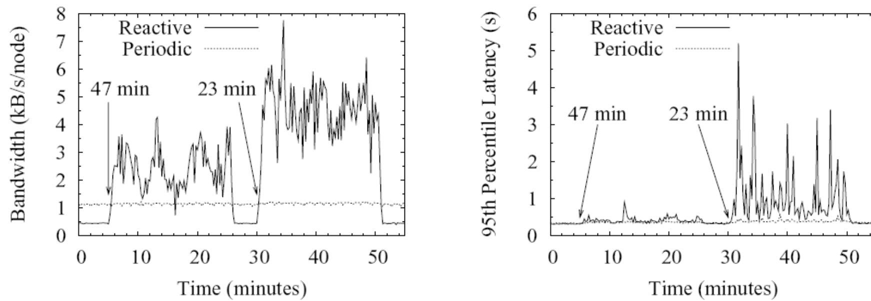


Figure 4.4: Reactive versus periodic recovery: bandwidth usage and latency [28].

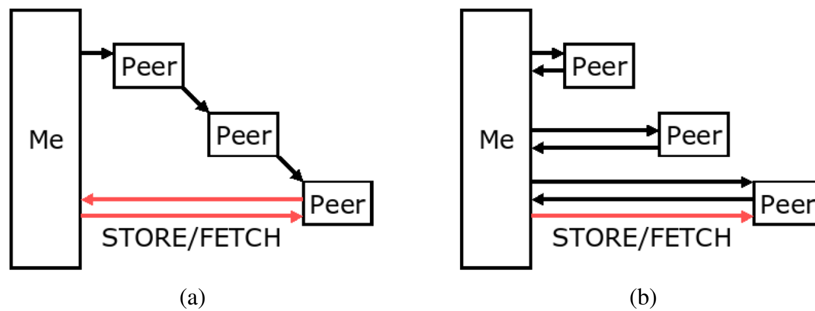


Figure 4.5: (a) Recursive versus (b) Iterative routing.

scheme which gives each node along the lookup path the opportunity to discover dead nodes in its routing table.

4.3.3 Security

The advantage of a structured P2P network like DHTs is the theoretical ease of efficient and effective lookups: keys can be found at nodes with identifiers closest to them with $O(\log n)$ messages. This structure, however, is at the same time a great security weakness.

In [18], the authors describe an *index poisoning* attack. The attack entails of inserting bogus entries in the index. In Overnet (a Kademlia based DHT), this can be done by inserting a (key, value) pair into the index, where the key is the hash of some targeted keyword and the value is a bogus random version identifier. Normally, the version identifier is the hash of a file which is used to find peers that have this file. The effect of this attack is that when a user searches for the targeted keyword, he will receive the bogus version identifier. If he selects this version identifier to download, the user's client will search indefinitely for sources that have this non-existing file. In a variation of the poisoning attack, the attacker first publishes a (key, version) pair for the targeted keyword and then publishes a (version, location) pair where the location is bogus, i.e. a non-present IP address.

```
def krpc_ping(self, id, _krpc_sender):
    sender = {'id' : id}
    sender['host'] = _krpc_sender[0]
    sender['port'] = _krpc_sender[1]
    n = self.Node().initWithDict(sender)
    self.insertNode(n, contacted=0)
    return {"id" : self.node.id}
```

Figure 4.6: The two highlighted lines show that when a Mainline DHT node receives a ping, it adds the sender to its routing table. Source: Mainline 5.2.2 [5].

Luckily, in the two BitTorrent DHTs, this form of poisoning has little impact. In these DHTs, only (swarm infohash, peer list) pairs are stored[2] and there are restrictions placed on what a node can store. A node can only insert its own address in a swarm's peer list by announcing he is participating in that swarm, not the address of someone else. The only possible malicious announces are those with a bogus swarm infohash. However, since the published infohash is bogus, no node would ever look it up. Hence, these malicious announces only pollute the DHT, consuming little bandwidth and memory at other nodes.

Still, DHTs are vulnerable to other types of attack. Malicious nodes can join the DHT and respond maliciously. For example, when a malicious node M receives a lookup message from a normal node A, it can reply that the requested key is available at a victim's host V. This is called a *redirection attack* and can be used to perform a DDoS attack on a victim. According to the authors of [34], this technique is not sufficient on its own to make the attack effective. The magnitude of the DDoS attack can be magnified using two techniques:

1. *Attraction:* In Kad, a malicious node can proactively push information about itself to a large number of nodes. This will force the contacted nodes to include the malicious node in their routing tables and thereby attracting more traffic to him.
2. *Multifake:* While attraction allows more nodes to be redirected to the victim, better amplification can be achieved by including the victim's IP address multiple times through the use of different logical identifiers, e.g. by using different port numbers.

While the authors focused on Kad, the amplification techniques can also be used in the BitTorrent DHTs. For example, Figure 4.6 shows the attraction vulnerability in the Mainline DHT. Using these different techniques, an attack of up to several Mbps can be generated by employing 5 malicious nodes in a Kad network (Figure 4.7). Traffic at each of the malicious nodes themselves did not exceed 450 Kbps. To make DHTs more resistant, it is necessary that an attacker's ability to

[2]Technically, the Azureus DHT also allows other information to be stored, but this is not relevant for the discussion at hand.

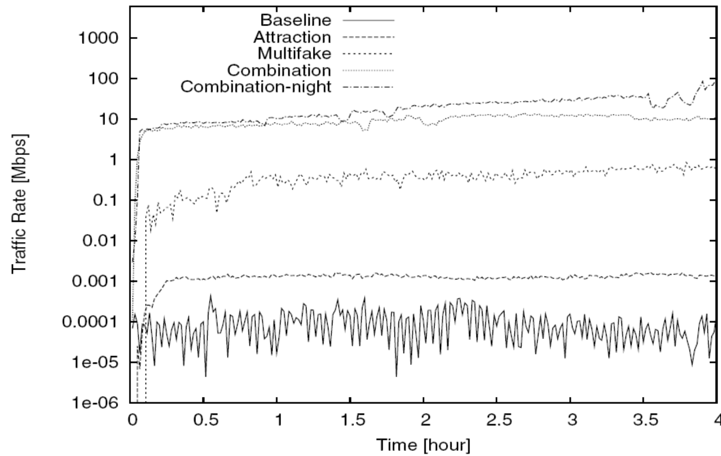


Figure 4.7: Traffic received by the victim caused by the DDoS attack initiated by 5 malicious nodes [34].

redirect or infect a large number of nodes is severely limited. A pull-based design is a possible solution to reduce the effectiveness of the attraction technique. The multi-fake technique can be rendered less effective by having bounding the communication from a node to a single physical address, but this may give rise to a *disconnection attack* [34]. It is also hard to prevent redirection attacks. It is important to validate whether a node C is actual part of the DHT, but doing so through a connection attempt is vulnerable for DDoSes as we have described. Delaying the membership validation until we have heard about node C from several other nodes seems like a good idea, but this is susceptible to Sybil Attacks [12].

4.3.4 Bootstrapping

The bootstrapping phase in a DHT requires that a node knows at least one other node that is already in the DHT. This other node is used to fill the routing table so it can operate normally.

The Mainline and Azureus clients differ in how they find such an initial DHT node. The Mainline client uses a DHT for swarms without a central tracker and finds the initial nodes' addresses in the corresponding torrent file. Azureus and Mainline-compatible clients, however, seem to use peers encountered in swarms as initial DHT nodes [11].

However, bootstrapping new nodes into the DHT overlay is not without problems. New nodes are often short-lived and immediately incorporating them into the DHT may not be a good idea. While no data is available on the Mainline DHT, Azureus seems to incorporate new nodes immediately, but their responsibility is limited through slow routing table updates. This can be seen in Figure 4.8: new nodes only receive few DHT messages in the first few hours, but load increases

steadily as routing tables are updated.

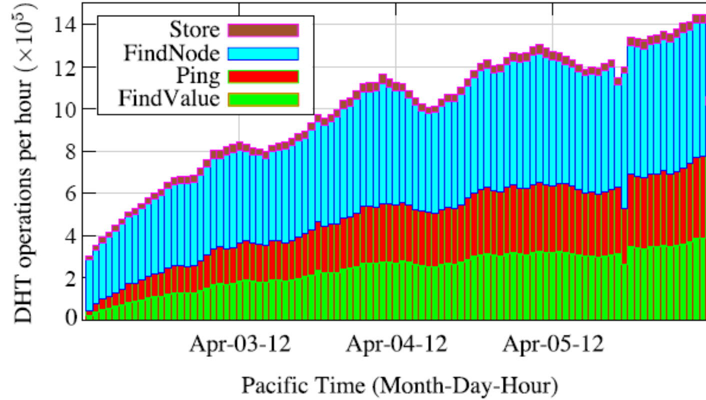


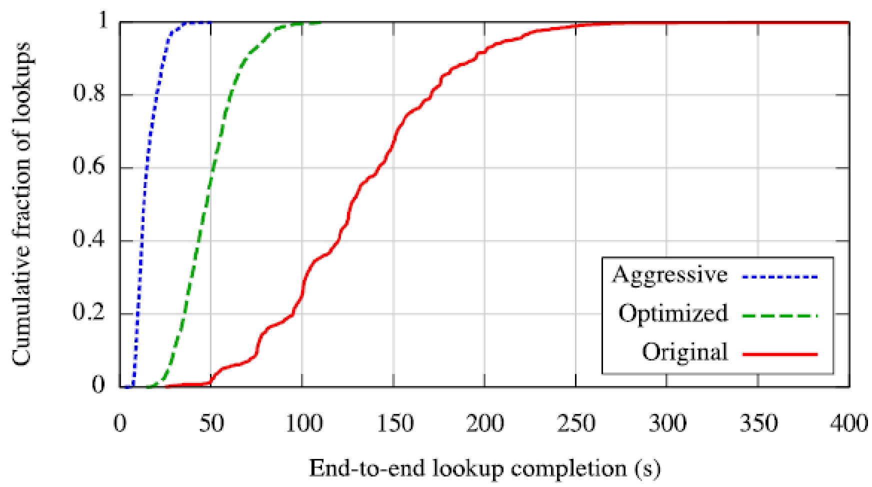
Figure 4.8: Two day trace of DHT messages received by 125 Azureus clients running simultaneously [14].

4.3.5 Load balancing and performance

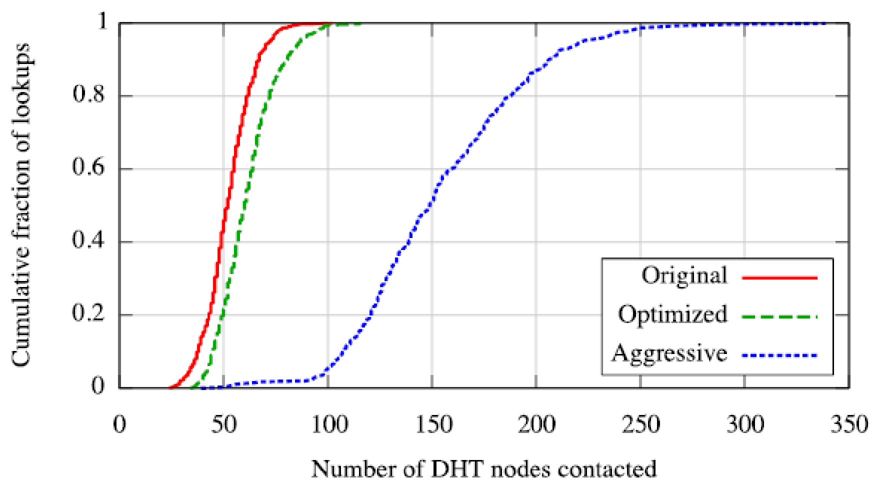
In a DHT, the node whose identifier is closest to a certain key is responsible for storing the value for that key. This can be problematic if the node has limited resources and the key is fairly popular. To mitigate this problem in the two BitTorrent DHTs, data is replicated among the k closest nodes known as replica nodes. For the Mainline DHT $k = 3$ and for the Azureus DHT $k = 20$. The Azureus DHT also has additional means for sudden increases of load, caused by e.g. flash crowds. When this happens, Azureus will migrate the stored values for a key in 10 alternate locations [11].

In both DHTs, the replica nodes are not explicitly identified as in DHTs like Pastry [31]. Instead, nodes in both DHTs use Kademlia's iterative lookup algorithm to find n nearest nodes to a given key. Nodes found so far are stored in a priority queue. The nearest nodes in the priority queue are iteratively queried until the n nearest nodes in the priority queue have all been contacted and no additional nearby nodes have been found [11].

But as briefly touched upon in Section 4.3.2, iterative lookups can be quite costly if some of the returned nodes seem to be dead. Without any optimizations, lookups can take more over a minute in the Mainline DHT or more than two minutes in the Azureus DHT on average [11]. Such high lookup times would be unacceptable if a DHT was used for live streaming applications such as the one presented in [22]. Only with aggressive optimizations by contacting many more nodes simultaneously in parallel, the lookup times can be reduced an order of magnitude (Figure 4.9).



(a)



(b)

Figure 4.9: End-to-end lookup performance in the Azureus DHT showing (a) completion times, and (b) number of nodes contacted for different strategies [14].

Chapter 5

Gossip Protocols and Graph Walks

In this chapter we introduce algorithms for exploring unstructured P2P networks to solve the swarm discovery problem, rather than structured networks as discussed in the previous chapter. In Section 5.1 we give a fuzzy definition of gossip protocols and explain two different ways of walking a graph. In Section 5.2 we show how gossip protocols and random walks can be used to solve the swarm discovery problem. The problems that may arise are discussed in Section 5.3.

5.1 Introduction to gossip protocols and graph walks

In an unstructured P2P network links between nodes are formed arbitrarily, unlike in a structured network. This makes joining an unstructured network easier, but makes lookups more complicated. We no longer know that a value with key k is stored at a certain node whose identifier is closest to k. Instead, information could be stored anywhere and thus nodes must perform some exploration.

This exploration can be done in several ways. Nodes can either disseminate information or wander around through the network. For the former, we will discuss gossip protocols in Section 5.1.1. For the latter, we will discuss random and ant walks in Section 5.1.2.

5.1.1 Gossip protocols

Gossip (or epidemic) protocols are probabilistic protocols for dissiminating information. Nodes perform a simple set of operations periodically and are not aware of the state of the entire system and thus solely act based on local knowledge [10]. The framework in Figure 5.1 is often used to define the set of gossip protocols: a peer selects another peer p using a given peer selection function, they exchange their states and finally they merge the state they received with their own local state.

```
# Active thread                    # Passive thread
while True:                        while True:
  wait(Δ)                            q, state_q = receive()
  p = selectPeer()                   send(q, mystate)
  send(p, mystate)                   mystate = update(state_q)
  state_p = receive(p)
  mystate = update(state_p)
```

Figure 5.1: Pseudocode of a gossip protocol framework.

However, this framework is too general as it covers nearly all message passing protocols. For reasons given in [10], the authors propose the use of the following intentional *fuzzy* feature list:

1. **random peer selection**
2. **only local information is available at all peers**
3. **round-based (periodic)**
4. **limited transmission and processing capacity per round**
5. **all peers run the same algorithm**

This feature list defines an idealized gossip protocol, allowing us to avoid discussions whether or not a protocol is a gossip protocol and let us focus on to which degree a protocol resembles the just presented ideal instead.

For the remainder of this subsection, we will focus on the peer selection function. Ideally, we would like this function to draw a uniform sample of *all nodes* in the network, but this is not feasible in practice since this would require that each node would have a complete membership table. So a decentralized scheme is needed to maintain membership information for gossip-based protocols. We need a *peer-sampling service* [17].

The authors of [17] propose that the peer-sampling service itself is based on a gossip paradigm: each node maintains a relatively small membership table that provides a partial view on the complete network, and refreshes this table periodically through gossips. Such a peer-sampling service is already available and heavily used in Tribler: BuddyCast [27].

5.1.2 Graph walks

Instead of gossipping, we can also search an unstructured network by walking randomly through it. Starting from a node, we randomly (possible biased) pick a neighbour and visit it. From this node, we repeat selecting and visting a random neighbour until we have reached some termination condition, e.g. we have visited n nodes. An example of random walk of length 4 is shown in Figure 5.2.[1]

[1]Note that we have described the *concept* of a random walk here. A possible way of *implementing* a random walk is by randomly forwarding a message at each node.

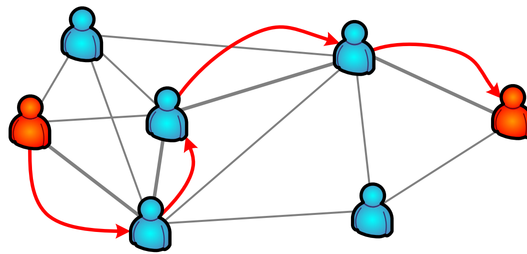


Figure 5.2: A random walk of 4 steps through a graph. At each node, a random link is taken.

Based on simulations, randomly walking through a graph is more efficient and effective than flooding, i.e. visiting all nodes within a certain radius, under certain conditions [16]. One of those conditions is in topologies with peer clustering. Peer clustering happens in e.g. the Tribler network where each Tribler client maintains a semantic overlay by staying connected to *taste buddies* [27].

A different way of walking through a graph is based on ants. Biological ants search their surroundings for food and leave pheromone traces when they have found some. Ants can smell these traces when they are nearby and follow them.

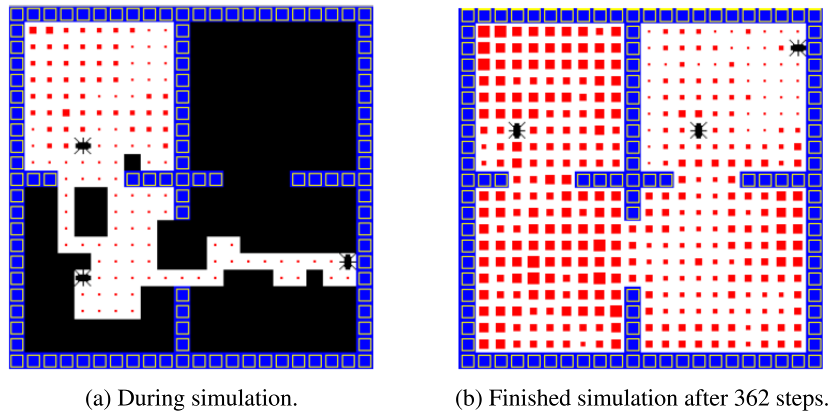
Introducing this biological concept to graphs, we will consider the *Vertex Ant Walk* (VAW) example from [44], an algorithm to cover all vertices in a graph. An artificial ant is able to leave pheromone traces on each vertex it visits. At each vertex it can "smell" the number of traces that have been left on that vertex and its adjacent neighbours, as well as the times of the most recent trace on each of them. We will use $\sigma(v)$ and $\tau(v)$ to denote the number of marks left and the time of the most recent mark on vertex v, respectively. $N(v)$ denotes the neighbouring vertices of vertex v.

```
def vertexAntWalk(u):
    v    = randmin(N(u), key = λx: (σ(x),τ(x))
    σ(u) = σ(v) + 1
    τ(u) = t
    t    = t + 1
    walkTo(v)
```

Figure 5.3: The algorithm for Vertex Ant Walk, executed by an ant at vertex u. The function `randmin()` selects the minimum value of a set, or picks one randomly when there are multiple candidates.

Figure 5.3 shows the algorithm that is executed by each ant at vertex u. Each ants sniffs around, looking for a neighbour v of u with the least number of marks on it, and in case of a tie, the oldest most recent mark. If there are multiple vertices with the same (σ, τ) pair, one is picked randomly. The ant will then move to vertex

v, but not before setting $\sigma(u)$ to $\sigma(v) + 1$ and updating $\tau(u)$. Notice and wonder why, that in VAW the ants are *avoiding* pheromone traces left behind instead of following them. A simulation of VAW can be seen in Figure 5.4.



(a) During simulation. (b) Finished simulation after 362 steps.

Figure 5.4: A simulation of the VAW algorithm with three ants. Black squares are cells that have not yet been visited. The size of a red square in cell u is proportional to $\sigma(u)$, i.e. the level of trace on u [44].

5.2 Discovering the swarm through gossips and walks

We will discuss two ways to perform BitTorrent swarm discovery based on techniques we just have introduced. We will discuss Peer Exchange (PEX), an already widely deployed gossip-based protocol, and swarm discovery using random walks. Both approaches assume that at least one member of the swarm is known.

5.2.1 Peer Exchange (PEX)

Peer Exchange is a BitTorrent extension designed to speed up swarm discovery [4, 25]. Instead of relying on peers supplied by a central tracker, peers can PEX with their neighbours, as shown graphically in Figure 5.5. After a peer has exchanged lists with another peer, it may connect to the newly discovered peers.

In more details, PEXing is done as follows. For each PEX-capable link, i.e. both endpoints support PEX, a peer keeps track of his own PEX set and the PEX set of its neighbour at the other end of the link, called the remote PEX set. These sets represent a (possibly partial) view on a peer's neighbourhood set and are used to send deltas. When a peer decides to send a PEX message, it uses its current neighbourhood set and its local PEX set to determine which connections have been added and which connections have been dropped since the last sent PEX message. At most 50 of each of these two deltas are sent to the other end of the link and not more often than once a minute [36].

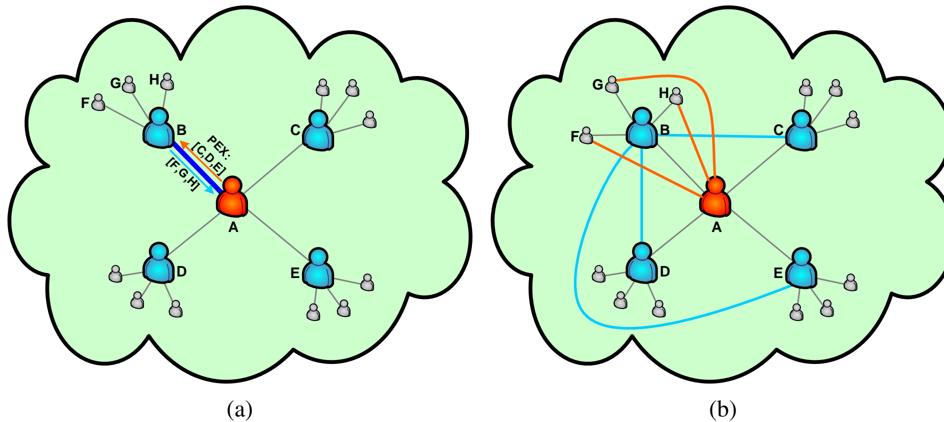


Figure 5.5: (a) Peer A and peer B are exchanging peer information. (b) Afterwards, peer A and peer B can connect to the newly discovered peers. Note: The two exchanged peer lists do not have to be sent in tandem.

From this description, PEX seems to be a gossip protocol, but it deviates from the ideal on two points:

1. Peers are not randomly selected to gossip with. Instead, peers gossip with *all* neighbours periodically.
2. PEX does not necessarily happen in tandem. According to the framework given in Figure 5.1, state from both peers are sent and received at the same time, but in the PEX protocol this is not enforced. Instead, a peer may choose its send rate independently.

5.2.2 Random walks

In [15], random walks are proposed for swarm discovery and even replacing the BitTorrent tracker. In this proposal, random walks are performed as described in Section 5.1.2 and are used to find a new neighbour to connect to by selecting the last visited node. Intuitively, the walk must be of a certain length if we want to randomly select a node from the graph. If the walks are too short, we would only find nearby nodes.

We can define a random walk algorithm by a $n \times n$ transition probability matrix P where each is row $\mathsf{P}_i \in [0, 1]^n$ is a distribution. A step of a random walk at node i samples a node j with probability P_{ij}. For an unbiased random walk in the graph shown in Figure 5.6, the corresponding matrix P is as follows:

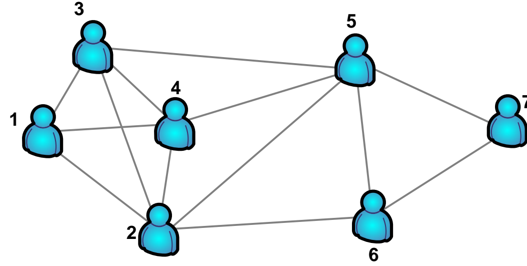


Figure 5.6: An example graph corresponding to the given matrix P.

$$\mathsf{P} = \begin{bmatrix} 0 & 0.3333 & 0.3333 & 0.3333 & 0 & 0 & 0 \\ 0.2000 & 0 & 0.2000 & 0.2000 & 0.2000 & 0.2000 & 0 \\ 0.2500 & 0.2500 & 0 & 0.2500 & 0 & 0 & 0 \\ 0.2500 & 0.2500 & 0.2500 & 0 & 0.2500 & 0 & 0 \\ 0 & 0.2000 & 0.2000 & 0.2000 & 0 & 0.2000 & 0.2000 \\ 0 & 0.3333 & 0 & 0 & 0.3333 & 0 & 0.3333 \\ 0 & 0 & 0 & 0 & 0.5000 & 0.5000 & 0 \end{bmatrix}$$

A so called *stationary distribution* is a distribution $\pi \in [0,1]^n$ that satisfies $\pi = \pi\mathsf{P}$. When such probability exists, the *mixing time* is the (minimum) number of t steps needed to obtain distribution π, such that the following holds for all initial distributions π_0, i.e. the distribution from the starting point of the walk [15]:

$$\pi \approx \pi_0\mathsf{P}^t$$

For some of the random walking algorithms covered in [15], the mixing time is known to be $t = O(\log n)$. For our example graph of $n = 7$ nodes, a mixing time of $t = \lceil \log_2 7 \rceil = 3$ seems to suffice:

$$\begin{aligned} \pi &= \pi\mathsf{P} \\ &\Leftrightarrow \{\pi \approx \pi_0\mathsf{P}^t\} \\ \pi_0\mathsf{P}^t &\approx \pi_0\mathsf{P}^t\mathsf{P} \\ &= \pi_0\mathsf{P}^{t+1} \\ &\Leftrightarrow \{\forall\pi_0 \Rightarrow \mathsf{P}\} \\ \mathsf{P}^{t+1} &\approx \mathsf{P}^{t+2} \\ \mathsf{P}^{3+2} - \mathsf{P}^{3+1} &= 10^{-3}. \end{aligned}$$

$$\begin{bmatrix} -0.0271 & 0.0036 & -0.0022 & -0.0022 & -0.0163 & -0.0063 & 0.0154 \\ 0.0060 & -0.0360 & -0.0078 & -0.0078 & 0.0103 & 0.0228 & -0.0233 \\ -0.0112 & -0.0013 & -0.0104 & -0.0056 & -0.0083 & -0.0030 & 0.0100 \\ 0.0040 & -0.0091 & -0.0181 & -0.0230 & 0.0154 & -0.0043 & -0.0027 \\ -0.0161 & 0.0046 & 0.0079 & 0.0079 & -0.0271 & -0.0140 & 0.0045 \\ -0.0052 & 0.0410 & -0.0026 & -0.0026 & -0.0298 & -0.0565 & 0.0211 \\ 0.0307 & -0.0521 & -0.0005 & -0.0005 & 0.0147 & 0.0296 & -0.0555 \end{bmatrix}$$

While this proposed algorithm can work fine on its own, the authors of [15] also propose the concept of *entry points*. These entry points are continuously walking the graph, allowing new peers to fill their initial neighbourset faster through them.

5.3 Problems encountered with gossips and walks

When using gossip protocols and graph walks for swarm discovery, we can encounter complicating factors described in Section 2.3. In this section we will discuss the problems that can arise.

5.3.1 Connectability

Connectability is an important issue when doing swarm discovery, which has not been tackled yet in the currently deployed PEX protocols. Early results from an extensive PEX performance measurement show that we can connect to only a small fraction of the peers we discover (Table 5.1). Only 16% to 25% of the known peers were connectable.

Time	14 Oct 16:16	14 Oct 17:16	15 Oct 10:16	16 Oct 14:00
Peers known	4562	4183	3782	3024
Peers connected	719 *16%*	470 *11%*	651 *17%*	743 *25%*
PEX sources	121	104	427*	514

Table 5.1: Results of a 3 day PEX crawl. On the second day, the crawler's code was changed, so more peers sent their PEX messages in [42].

A plausible explanation is the following. Information received through PEX only tells us to which peers our neighbour is connected (assuming our neighbour is not malicious). We do not know whether each of those connections is a local (outgoing) or a remote (incoming) connection. Outgoing connections are not a problem, since this means the other endpoint is connectable, but a peer that initiated the connection to our neighbour may not be connectable itself.

The same problem can arise when using random walks. The peers we discover with random walks are discovered by walking over already established connections. So when we found a peer, we do not know whether it is connectable or not.

It is therefore important that peers should check themselves whether they are connectable and make this information available to others. Additionally, a peer should only inform others of the existence of a newly discovered peer if the discovered peer is known to be connectable. In case of PEX this means a peer should only inform other PEX-capable peers about its outgoing connections. The only problem that then remains is when a peer becomes unconnectable later on because of congestion or high load.

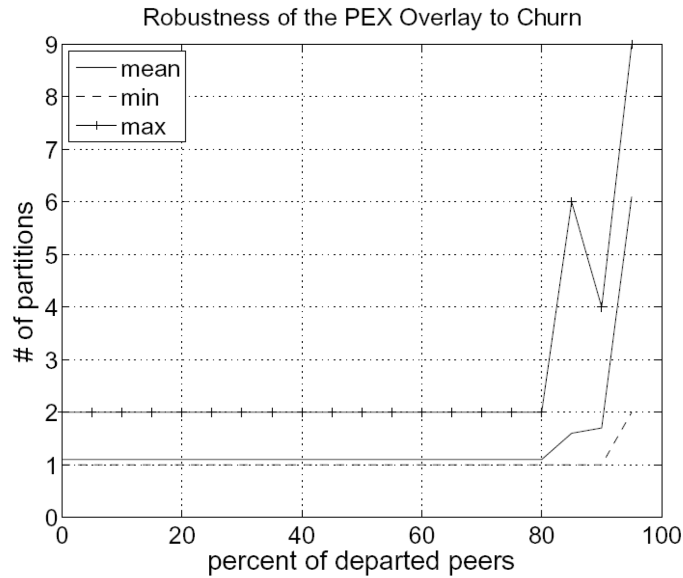


Figure 5.7: Impact of churn on the PEX overlay constructed using both a tracker and PEX [1].

5.3.2 Churn

The problem of churn is twofold: it could possibly cause a graph become disconnected and it causes membership information of the network to become stale.

The first problem seems not to appear with PEX-constructed overlays. When up to 80% of the peers departed in a simulation presented in [1], at most two disconnected subgraphs occurred, with the smaller subgraph containing a single peer. This simulation assumed the availability of both a central tracker and PEX-enabled peers (Figure 5.7).

Also, when the only available way to discover the swarm is PEX, swarm fragmentation is unlikely. The clustering coefficient [45] in the constructed overlays range from 6% to 15% according to a simulation done by Victor Grishchenko (Table 5.2). Considering real-world peers usually have around 50 connections and a list of unconnected addresses, the likelihood of a PEX-only swarm getting fragmented is small.

Regarding the staleness of PEX data, results of the PEX crawl measurement mentioned in Section 5.3.1 showed:

- Intersection between 1st run and 2rd run connected peers (719 and 470, respectively) is still 73 peers.
- Intersection between 1st run and 3rd run connected peers (719 and 651, respectively) is only 15 peers.
- Intersection between 2nd run and 3rd run connected peers (479 and 651, respectively) is only 10 peers.

	Scenario	Clustering Coefficient
I	10,000 peers start being randomly interconnected, 40 connections per peer on average. Each step, a peer with more than 20 connections disconnects from a random neighbour. A peer with at most 20 connections connects to a random neighbour of a random neighbour.	Stabilizes at 15%
II	Same as Scenario I, except a peer connects to a random peer in the set of its neighbours' neighbours (common acquantainces among neighbours in Scenario I had a higher probability of getting chosen).	Stabilizes at 6%
III	Random peer removal added to Scenario I: At each step, one peer is completely rewired.	Stabilizes at 12-14%

Table 5.2: Simulations of PEX-only. Source: Victor Grishchenko.

- The lists of the first three runs intersect by just 3 peers [42].

This means that old PEX data is highly unreliable and only live PEX data is to be preferred.

5.3.3 Security

Gossip-based protocols are vulnerable to false membership information. Malicious nodes can falsely inform others that node V is part of the network. Using multifake (Section 4.3.3), attacks of several Mbps can be generated with just a few percent of the network being malicious [34].

In the PEX protocol, exploiting this vulnerability is limited, although no experimental data is available as far as we know. PEX limits the number of new peer addresses being sent per minute to 50, and BitTorrent clients like Azureus [2] only connect to the same physical IP address per swarm once by default. Nevertheless, the protocol can be made more resistant by delaying membership validation [34].

5.3.4 Bootstrapping

PEX is currently used in conjunction with a central tracker. Once at least a peer in the swarm has been found, PEXing can begin.

The initial peers, however, can also be found by carrying state across multiple sessions, even though a large portion of previously known peers might be no longer online. For example, the BitTorrent client Azureus [2] is known to save peer addresses to disk in order to bootstrap faster into previously joined swarms. Tribler [40] takes it a step further and stores additional information in its MegaCache, remembering how good previously encountered peers were, their tastes, and more.

Chapter 6

Proposed Distributed Tracking Algorithm

In this chapter, we propose a first version of a fully distributed solution to the swarm discovery problem. In Section 6.1, we will describe the environment requirements for a swarm discovery algorithm. In Section 6.2, we define and explain our proposed algorithm: 2-Hop TorrentSmell.

6.1 Environment

The presented algorithm in this chapter is designed to be deployed within Tribler and thus requires the availability of a peer-sampling service such as *BuddyCast* (see Section 5.1.1). It is required that it maintains a live overlay with at least 10 peers it encountered using the peer-sampling service. The algorithm also requires that PEX support is available (5.2.1) and that each client can store information across sessions. That is, a client should be stateful rather forgetful and use something similar to Tribler's *MegaCache*.

Additionally, this version of the algorithm assumes that swarms already exist and were created through other existing means, e.g. a central tracker or BitTorrent's DHT extension. It does not support the creation of swarms yet.

6.2 2-Hop TorrentSmell

Our distributed solution to the swarm discovery problem is built upon the idea that peers stay connected to the last N swarms he downloaded from and keeps tracking of them using PEX. The solution uses a gossip protocol to find these PEXing peers. We will now describe the algorithm in full details in a step-by-step manner. Figure 6.1 to Figure 6.6 correspond to step 1 to 6, respectively.

1. Each peer stays connected to the last $N = 25$ swarms he downloaded from and continuously runs the PEX protocol to discover the swarm. The choice

of $N = 25$ is arbitrary and further research is needed to determine a feasible and effective value of N.

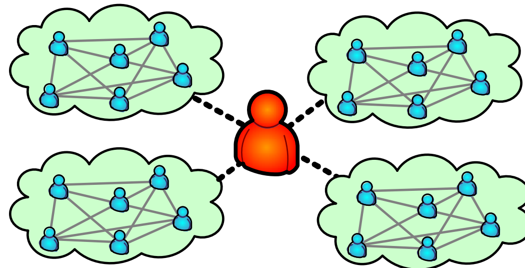


Figure 6.1: Each peer PEXes the last 25 swarms.

2. Peers gossip the last 25 swarms they have downloaded from using the BuddyCast gossip protocol.

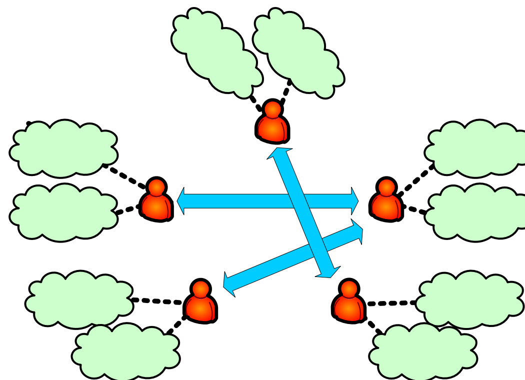


Figure 6.2: Peers gossip about the last 25 swarms they have downloaded from.

3. Tribler peers store the information they receive via BuddyCast messages in their MegaCache for future searches. For each swarm they store the SHA1 infohash, the filename of its content and which peers claim they have downloaded that file. For each peer they remember when it was last seen and what its IP address was.

4. When the user wants to download a file, it enters a keyword. For example, it issues the query “obama”. This query is sent to 10 connected peers. It also searches its own MegaCache.

MegaCache

Swarm Table

Swarm	Downloaded by
Swarm1 SHA1 / Filename	Peer3 Peer4 ...
Swarm2 SHA1 / Filename	Peer19 Peer51 Peer354 ...
...	...

Peer Table

Peer	IP	Last Seen
...	...	...
...	...	...
...	...	...
...	...	...
...	...	...
...	...	...

Figure 6.3: Peers store information they receive in their MegaCache.

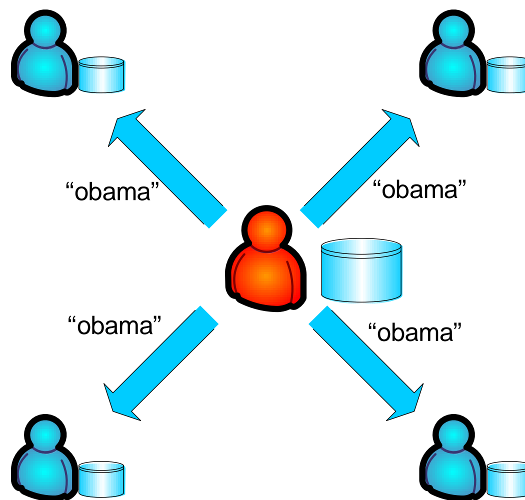


Figure 6.4: A user issues a query, e.g. "obama" to 10 connected peers.

5. The peers respond to the issued query by returning matching swarm names and the freshest entries from their peer table.

6. The user's client merges the results and only connects to 20 peers appearing in multiple results as an anti-DDoS measure. The user's client can then switch to PEX to perform further swarm discovery.

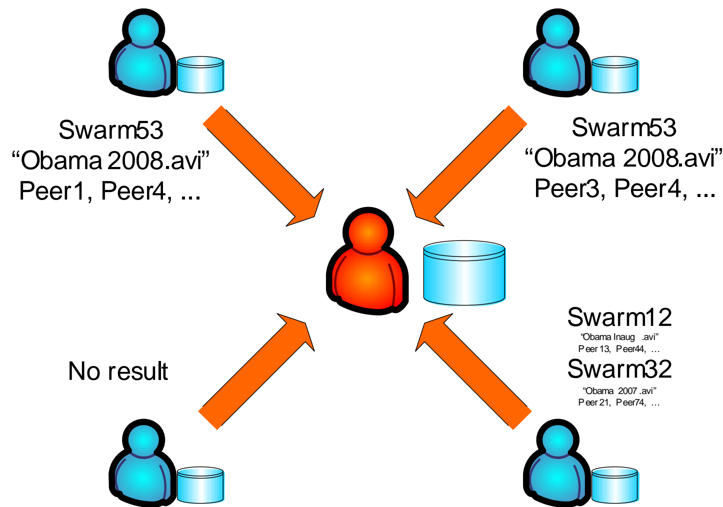


Figure 6.5: The 10 connected peers respond to the query by returning matching swarm names and the freshest, corresponding peers.

Swarm53, “Obama 2008.avi”

Peer 1	Peer 3	Peer 4
Peer 4	Peer 4	Peer 6
Peer 6	Peer 9	...
...	...	...

Figure 6.6: The received results are merged and only 20 peers appearing in multiple results are connected to. In this example, peer 4 and 6 would be possible candidates.

Chapter 7

Conclusions

In this chapter we will summarize our motivation and conclusions of our discussion of the problems presented in this report. We will then conclude this report with further research that remains on the subject of swarm discovery.

7.1 Conclusions

Peer-to-peer applications are popular and will become more important in the future. Forecasts show a move from traditional television to the online equivalents. As P2P will play an increasingly larger role in our lives, we must make sure the next generation P2P software will be fault tolerant, secure and know no limits to scalability. For this, it is essential that no central component remains.

Swarm discovery is the process of finding interesting peers and is a crucial part of the P2P system BitTorrent. Solving the problem of swarm discovery is complicated by dynamic aspects as connectability, churn, security and bootstrapping. Absolute care is needed to make sure that a solution is efficient, effective and secure.

We have found that a large portion of public swarms nowadays are tracked by a single group of trackers. Trackers are the central component in the BitTorrent protocol solving the swarm discovery problem and are thus BitTorrent's Achilles heel. If this large tracker farm was to fail, an important source for finding peers for numerous BitTorrent swams disappears.

We have seen that Distributed Hash Tables are in use to solve the swarm discovery problem in a decentralized way, but fails to do so in a secure and efficient way. Their structure makes it easy to be exploited, allowing for Distributed Denial of Service attacks of great magnitude, and their lookup completion times can run up to a few minutes, which is rather unacceptable for finding initial peers at the start of a download.

Better distributed alternatives are to be found in gossip-based protocols such as PEX, and random walks. These classes of algorithms work in unstructured networks, which withstand churn easily. Security can still be an issue, but there are

well-known solutions. Connection problems, however, seem to be more prominent in the currently deployed solutions than in the deployed DHT, although they can be largely prevented if peers check their own connectivity.

Based on the studied problems and solutions, we have proposed our 2-Hop TorrentSmell algorithm. This algorithm makes use of two gossip protocols to solve the swarm discovery problem. It relies on peers staying online in swarms and continuously discovering it using the PEX protocol, and uses the epidemic BuddyCast protocol to find these PEXing peers. This algorithm is a first step towards a fully distributed BitTorrent.

7.2 Further research

In the subsequent Master of Science project, we will implement and deploy our proposed 2-Hop TorrentSmell algorithm in Tribler. We will study its effectiveness, security and performance, either through simulation or a real world test on existing large swarms.

We might also need to research the effectiveness of PEX and design a solution so that PEXing peers can provide peer lists of the best possible quality. The reason is that foreign PEX data often seems to contain many unconnectable peers. A possible step forward could be to extend PEX messages in such a way that Tribler peers can inform each other about the type of each connection (local or remote) in the sent PEX messages.

Further research that falls outside the scope of the described project involves eliminating the central tracker completely. Our currently proposed algorithm is only a first step in this direction, since it assumes swarms already exists. We do not have not a solution yet to the chicken-and-egg problem: how to create swarms and track them in a distributed manner.

Bibliography

[1] A. Al-Hamra, A. Legout, and C. Barakat. Understanding the Properties of the BitTorrent Overlay. Technical report, INRIA, 2007.

[2] Azureus – BitTorrent client. http://azureus.sourceforge.net/.

[3] AzureusWiki: Distributed hash table. http://www.azureuswiki.com/index.php/Distributed_hash_table.

[4] AzureusWiki: Peer Exchange. http://www.azureuswiki.com/index.php/Peer_Exchange.

[5] BitTorrent – The Mainline BitTorrent client. http://download.bittorrent.com/dl/archive/.

[6] BitTorrent.org: DHT Protocol, 2008. http://www.bittorrent.org/beps/bep_0005.html.

[7] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. *SIGOPS Operating Systems Review*, 36(SI):299–314, 2002.

[8] Cisco Systems, Inc. Approaching the Zetabyte Era. http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481374.html, June 2008.

[9] B. Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, volume 6. Berkeley, CA, USA, 2003.

[10] P. Costa, V. Gramoli, M. Jelasity, G.P. Jesi, E. Le Merrer, A. Montresor, and L. Querzoni. Exploring the interdisciplinary connections of gossip-based systems. *SIGOPS Operating Systems Review*, 41(5):51–60, 2007.

[11] S. A. Crosby and D. S. Wallach. An Analysis of BitTorrent's Two Kademlia-Based DHTs. Technical Report TR-07-04, Rice University, June 2007.

[12] J. R. Douceur. The sybil attack. In *Peer-To-Peer Systems: First International Workshop, Iptps 2002, Cambridge, Ma, USA, March 7-8, 2002, Revised Papers*, page 251. Springer, 2002.

[13] CIA Factbook. The World Factbook. https://www.cia.gov/library/publications/the-world-factbook/, 2008.

[14] J. Falkner, M. Piatek, J. P. John, A. Krishnamurthy, and T. Anderson. Profiling a Million User DHT. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 129–134. ACM New York, NY, USA, 2007.

[15] C.P. Fry and M.K. Reiter. Really Truly Trackerless BitTorrent. *School of Computer Science, Carnegie Mellon University, Tech. Rep*, pages 06–148, 2006.

[16] C. Gkantsidis, M. Mihail, and A. Saberi. Random walks in peer-to-peer networks. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 1, 2004.

[17] M. Jelasity, S. Voulgaris, R. Guerraoui, A.M. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3), 2007.

[18] J. Liang, N. Naoumov, and K. W. Ross. The Index Poisoning Attack in P2P File Sharing Systems. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–12, 2006.

[19] K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys & Tutorials, IEEE*, pages 72–93, 2005.

[20] Marketing Charts. Internet Surpasses TV in Australia, Mobile Approaches Saturation Point.
`http://www.marketingcharts.com/television/internet-surpasses-tv-in-australia-mobile-approaches-saturation-point-3976/`, March 2008.

[21] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. *Proceedings of IPTPS02, Cambridge, USA*, 1:2–2, 2002.

[22] J. J. D. Mol, A. Bakker, J. A. Pouwelse, D. H. J. Epema, and H. J. Sips. The Design and Deployment of a BitTorrent Live Video Streaming Solution. I-Share Deliverable 1.21, `http://www.cs.vu.nl/ishare/`.

[23] NationMaster. Televisions by country. `http://www.nationmaster.com/graph/med_tel-media-televisions`. Data based on The CIA World Factbook, 2003.

[24] NewTeeVee. Online TV Watching to Grow, But Not So Fast.
`http://newteevee.com/2008/04/04/online-tv-watching-to-grow-but-not-so-fast/`, April 2008.

[25] A. Norberg and L. Strigeus. libtorrent: extension protocol.
`http://www.rasterbar.com/products/libtorrent/extension_protocol.html`.

[26] J. A. Pouwelse, P. Garbacki, D. Epema, and H. J. Sips. A Measurement Study of the BitTorrent Peer-to-Peer File-Sharing System. Technical Report PDS-2004-007, Delft University of Technology, 2004. ISSN 1387-2109.

[27] J. A. Pouwelse, J. Yang, M. Meulpolder, D. H. J. Epema, and H. J. Sips. Buddycast: an Operational Peer-to-Peer Epidemic Protocol Stack. In G. J. M. Smit, D. H. J. Epema, and M. S. Lew, editors, *Proc. of the 14th Annual Conf. of the Advanced School for Computing and Imaging*, pages 200–205. ASCI, 2008.

[28] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference*, 2004.

[29] Rixstep. TPB Raking in Millions.
`http://rixstep.com/1/20060708,00.shtml`, July 2006.

[30] J. Roozenburg. Secure Decentralized Swarm Discovery in Tribler. MSc thesis, Delft University of Technology, November 2006.

[31] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Lecture Notes In Computer Science*, 2218:329–350, 2001.

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

[33] D. Stutzbach and R. Rejaie. Understanding Churn in Peer-to-Peer Networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 189–202. ACM New York, NY, USA, 2006.

[34] X. Sun, R. Torres, and S. Rao. DDoS Attacks by Subverting Membership Management in P2P Systems. In *Secure Network Protocols, 2007. NPSec 2007. 3rd IEEE Workshop on*, pages 1–6, 2007.

[35] The Pirate Bay – BitTorrent Tracker. http://thepiratebay.org.

[36] TheoryOrg: BitTorrent Peer Exchange Conventions. http://wiki.theory.org/BitTorrentPeerExchangeConventions.

[37] TorrentFreak. 50% Of All BitTorrent Downloads are TV-Shows. http://torrentfreak.com/50-percent-bittorrent-downloads-tv-080214/, February 2008.

[38] TorrentFreak. News from The Pirate Bay Press Conference. http://torrentfreak.com/news-from-the-pirate-bay-press-conference-090215/, February 2009.

[39] TorrentFreak. Top 10 Most Pirated TV Shows on BitTorrent. http://torrentfreak.com/top-10-most-pirated-tv-shows-on-bittorrent-090220/, February 2009.

[40] Tribler – BitTorrent client. http://www.tribler.org.

[41] Tribler Wiki: 4th Generation of P2P. https://www.tribler.org/trac/wiki/4thGenerationP2P.

[42] Tribler Wiki: PEXCrawl. https://www.tribler.org/trac/wiki/PEXCrawl.

[43] Tweakers.net. Suprnova en andere torrentsites stoppen definitief. http://tweakers.mobi/nieuws/35467/, December 2004.

[44] I.A. Wagner, M. Lindenbaum, and A.M. Bruckstein. Efficiently searching a graph by a smell-oriented vertex process. *Annals of Mathematics and Artificial Intelligence*, 24(1):211–223, 1998.

[45] Wikipedia: Clustering coefficient. http://en.wikipedia.org/wiki/Clustering_coefficient.