

Distributed Tracking: 2-Hop TorrentSmell (TODO)

Raynor Vliendhart



Delft University of Technology

Distributed Tracking: 2-Hop TorrentSmell (TODO)

Master's Thesis in Computer Science

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

Raynor Vliendhart

16th February 2010

Author

Raynor Vliegendhart

Title

Distributed Tracking: 2-Hop TorrentSmell (TODO)

MSc presentation

TODO GRADUATION DATE

Graduation Committee

TODO GRADUATION COMMITTEE Delft University of Technology

Abstract

TODO ABSTRACT

Preface

[TODO] TODO MOTIVATION FOR RESEARCH TOPIC

TODO ACKNOWLEDGEMENTS

Raynor Vliegendhart

Delft, The Netherlands
16th February 2010

Contents

Preface	v
1 Introduction	1
1.1 BitTorrent	2
1.2 Tribler	2
1.3 Contributions	2
1.4 Thesis outline	3
2 Problem Definition	5
2.1 Swarm discovery	5
2.2 System requirements	6
2.3 Existing solutions	6
2.4 Problem aspects	7
2.4.1 Connectability	7
2.4.2 Churn	7
2.4.3 Security	8
2.4.4 Bootstrapping	8
3 PEX Behaviour Study	11
3.1 Experimental setup	11
3.2 Measurement results	11
3.2.1 Visualisation of the crawled swarms	12
3.2.2 Differences in PEX implementations and quality	18
3.2.3 Remaining peer uptime	20
4 2-Hop TorrentSmell Design	25
4.1 Some section	25
4.2 Remote Content Search	25
4.3 Tracking the swarm using RePEX	25
4.3.1 Used terminology	25
4.3.2 RePEX algorithm	26
4.3.3 Current Implementation	28

5	Implementation and Experiments	31
5.1	Cost of RePEX	31
6	Deployment	33
6.1	Foo	33
7	Conclusions and Future Work	35
7.1	Conclusions	35
7.2	Future Work	35

Chapter 1

Introduction

Peer-to-peer (P2P) applications are becoming increasingly more popular, and rightly so (Figure 1.1). The distributed nature of P2P technology allows us to easily and cost-effectively distribute a wealth of content all over the world. Highly popular content will be requested and downloaded by interested nodes in a P2P network, resulting in more available replicas.

Unfortunately, building P2P applications in a fully distributed way is not easy. The past has shown that designers often choose the easy way out and sacrifice full distribution by introducing central components in their systems. These central components prevent a P2P system to become fully self-organised and unboundedly scalable. When a central component becomes overloaded, breaks down or is taken down, the whole system suffers. A prime example is the legal case against Napster, where it was forced to shut down its centralized search facility [9].

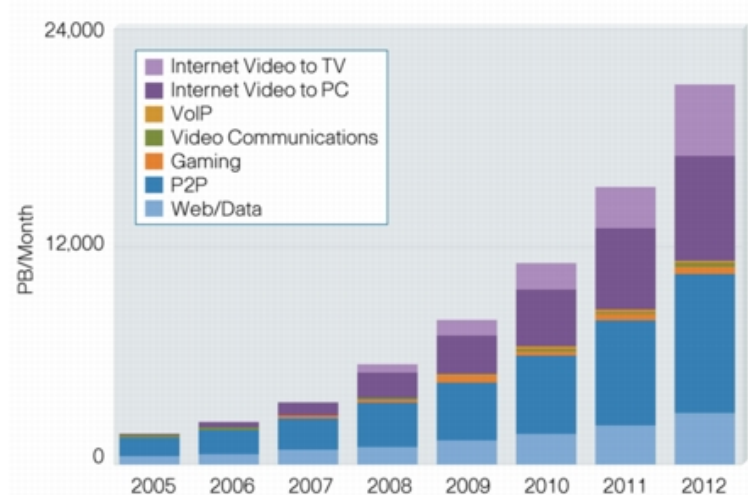


Figure 1.1: Global consumer Internet traffic forecast. Source: Cisco, 2008 [4].

Removing the need for any central component or server is a key aspect of the 4th Generation of P2P [17] and is the focus of the Tribler research project. Tribler is a P2P application based on BitTorrent developed by Delft University of Technology and Vrije Universiteit Amsterdam [16] and is part of the P2P-Next project funded by the European Union [10]. In this thesis, we will design and implement a distributed solution to finding peers for Tribler, replacing the central tracker component of the underlying BitTorrent protocol.

The remainder of this introductory chapter. . .

1.1 BitTorrent

BitTorrent is a file sharing P2P protocol created by Bram Cohen in 2001 [5]. Unlike previous P2P systems, BitTorrent peers downloading the same file can benefit from each other by exchanging pieces of the same file. This is called *bartering*. Bartering makes file distribution scalable. More downloaders means more copies of file pieces are available, and hence, more peers to barter with. Details on bartering can be found in [6].

To download a file through BitTorrent, a peer first needs a *torrent file* containing *metadata*. The metadata consists of the file name, file size, integrity hashes, and the URL of one or more *trackers* [6]. A tracker is a central component responsible for tracking peers in a *swarm*. A swarm is the total group of downloaders of a certain file and is identified by a hash derived from the metadata. A peer regularly contacts the tracker to announce its presence and receive a list of peers for a certain swarm. It uses this list of peers to find new peers to barter with.

1.2 Tribler

Tribler [TODO]

- PermID
- Tribler Overlay
- BuddyCast
- Remote Content Search

1.3 Contributions

In this thesis, we make the following contributions:

- [TODO:] something about PEX behaviour?
- [TODO:] A distributed tracking algorithm called 2-Hop TorrentSmell.
- [TODO:] etc.?

1.4 Thesis outline

The remainder of this thesis is organized as follows. Chapter 2 [etc...]

Chapter 2

Problem Definition

The following problem is the main subject of this thesis: 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. [TODO:] We give a short overview of the current deployed solutions to this problem in Section 2.3. Solving the problem in a distributed setting is complicated by dynamic aspects, which we cover in Section 2.4.

2.1 Swarm discovery

[Note:] This section is still a copy/paste. Should we still use the term “swarm discovery”? Should we call the problem “distributed tracking” instead? (Although distributed tracking would actually only reflect the RePEX part of 2-Hop TorrentSmell, not the Extended Remote Content Search)

[Use simple def: “Goal is finding peers to download content from”]

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 [12], *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 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.

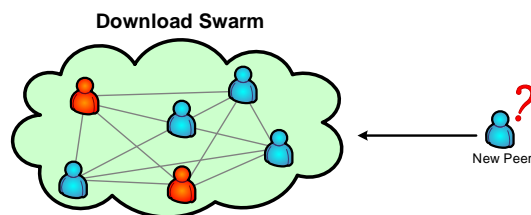


Figure 2.1: Swarm discovery is the problem of finding nearly all peers in a specific swarm. In the depicted situation, the new peer must somehow acquire contact information of the peers in the download swarm.

2.2 System requirements

- Performance: latency/bandwidth
- Scalability
- Security
- Etc.?

[THE FOLLOWING IS STILL A COPY/PASTE FROM RESEARCH ASSIGNMENT]

2.3 Existing solutions

[“No solutions exist which fully adheres to all requirements. . .”]

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 [6]. The process of joining the swarm is shown in Figure 2.2.

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 [2, 3], which we discuss in Chapter [TODO].

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 [TODO].

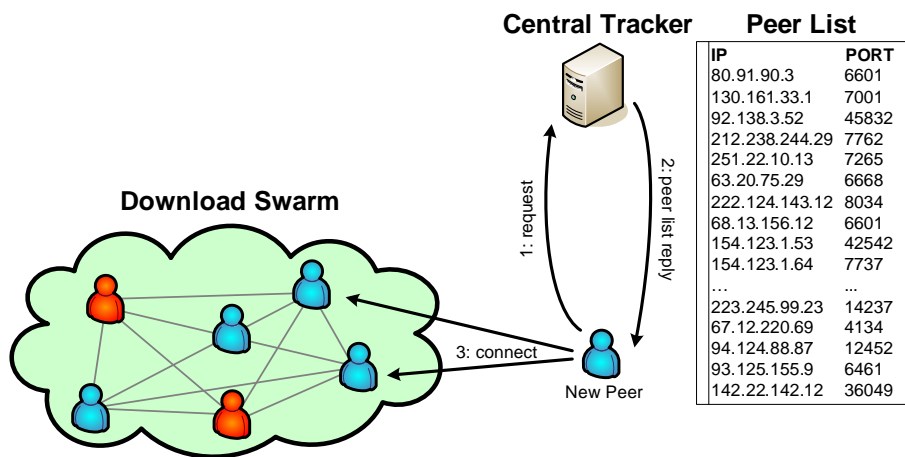


Figure 2.2: The swarm discovery problem can be solved using a central server called a tracker. Here, a new peer uses the tracker to find peers to connect to. [Use short sentence here]

2.4 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.

2.4.1 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 [11]. 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.

2.4.2 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 [13]. 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 constitute 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.

2.4.3 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 [8]. 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 [14].

2.4.4 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.

¹For example, Azureus' DHT consists of more than 1 million nodes [7].

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.

[END OF COPY/PASTE]

Chapter 3

PEX Behaviour Study

We have conducted a study to understand the reliability and usability of PEX messages. In this chapter we present the results of this study.

Section 3.1 describes the crawler software we have developed for our experiments. In Section 3.2 we analyse the data that we have acquired from two experiments we have performed.

3.1 Experimental setup

We study the behaviour of PEX by crawling through BitTorrent swarms. Our crawler connects to peers in a BitTorrent swarm and waits idly for an incoming PEX message. When the crawler receives a PEX message, it connects to all the peers listed in the message which it has not seen before. When a peer does not send a PEX message within a reasonable time (2 minutes), or it does not support PEX at all, our crawler closes its connection to that peer and moves on. The crawler uses a central tracker to find initial peers to connect to.

During the crawl, the crawler logs when it connected to a peer and that peer's version, when it received a PEX message and all peers listed within, and when it closed the connection.

In addition to crawling through the swarm, our crawler also regularly checks whether a previously encountered peer still responds to incoming BitTorrent connection requests. This information, dubbed "online check" is also logged and should give use an idea of the churn in a swarm.

3.2 Measurement results

Using our crawler software, we have performed two crawls. We crawled three Linux distribution swarms (Ubuntu, Gentoo and Fedora) in early September and the ten largest swarms of a large public tracker in late September. The first crawl has run for 2 hours, the second crawl has run for 20 hours.

In Section 3.2.1, we visualize the swarms we have crawled. In Section 3.2.2, we analyse properties of PEX messages. Section 3.2.3 focuses on the churn in the swarms we have crawled.

3.2.1 Visualisation of the crawled swarms

During the crawls, we noticed that our crawler quickly discovered new peers and connected to them. The number of peers found and connected as a function of time is shown in Figure 3.1. These graphs also show that the crawler never manages to connect to all peers it has discovered via its initial contact with the trackers and through subsequent PEX messages. Notice that both graphs roughly cover a time span of 12 minutes and 10 hours, respectively, while the crawler in both instances ran much longer than that. This fact excludes the possibility the crawler did not have enough time to connect to all discovered peers. Hence, a majority of discovered peers are unresponsive.

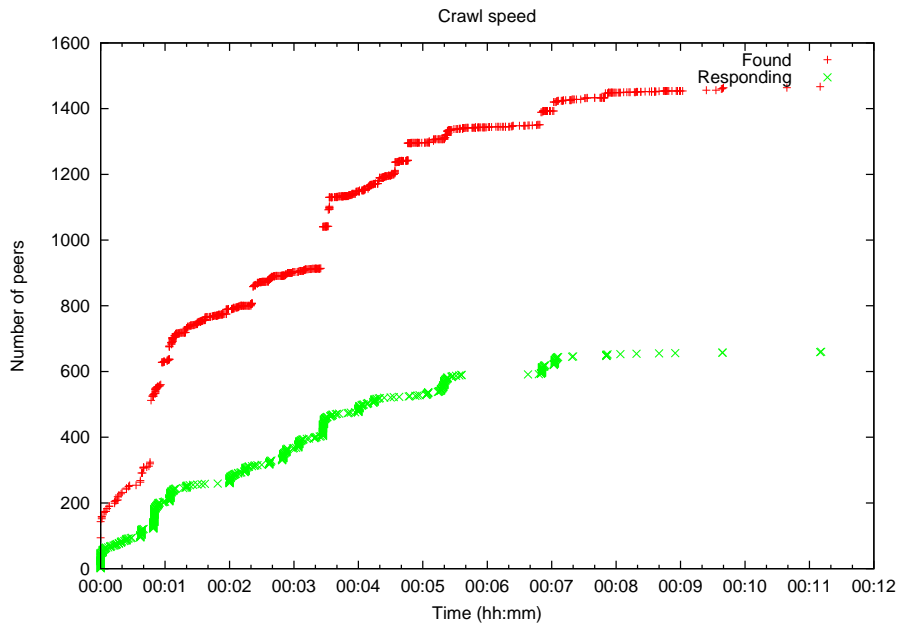
To investigate the nature of the unresponsive peers, we have visualized the Ubuntu swarm as a graph in Figure 3.2. We have omitted the other swarms as they were either too small to be interesting or too large to be visualized in this way. The graph of the Ubuntu swarm shows a large central cluster located to the north of the tracker (represented by a blue square). The amount of unresponsive peers in this cluster is not remarkable. These peers could be firewalled or congested.

More interestingly are two small separated clusters. One is located left of the central cluster (Figure 3.3a), and the other is at the bottom of the graph (Figure 3.3b.) The crawl logs tell us that the left cluster consists of Transmission BitTorrent clients. The unresponsive peers they are allegedly connected with all have an IP in the 0.x.x.x, 1.x.x.x, or 2.x.x.x range.¹ These address ranges were either private or unallocated at the time of the crawls [1] and are useless for other peers. This means the PEX implementation of certain Transmission versions is faulty or buggy. The crawl logs indicate that at least versions 1.74 and 1.75b2 seem to be affected.

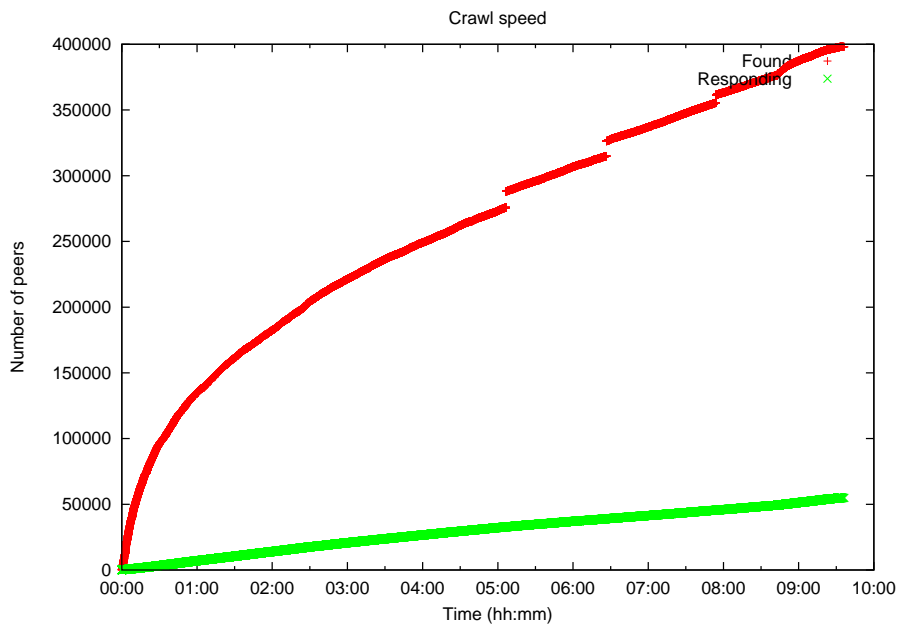
The bottom cluster consists of Deluge clients. Contrary to the unresponsive peers in the Transmission cluster, these unresponsive peers do have valid public IP addresses and reside in various subnets. So in theory, if these peers are not firewalled, they should be connectible. Oddly enough, these peers are only reported in PEX messages sent by the Deluge peers. Even if these peers were firewalled, they should at least be able to connect to other peers in the swarm and eventually show up in PEX messages sent by non-Deluge peers, yet mysteriously they do not. It is hard to guess what is going on here.

But do the faulty PEX implementation of the Transmission clients and the behaviour of these peculiar Deluge clients have a large impact? Figure 3.4 shows that Deluge (-DE) is not very common and that Transmission (-TR) has a moderately sized market share. More worryingly might be the existence of clients that seem to support PEX, yet never seem to send to any PEX messages, e.g. Azureus (-AZ).

¹Peers with a 0.x.x.x IP address were not drawn to reduce visual clutter.



(a) Linux crawl (3 swarms)



(b) Public tracker crawl (10 largest swarms)

Figure 3.1: Number of peers found (via tracker and through PEX) and responding (connected) as a function of time.

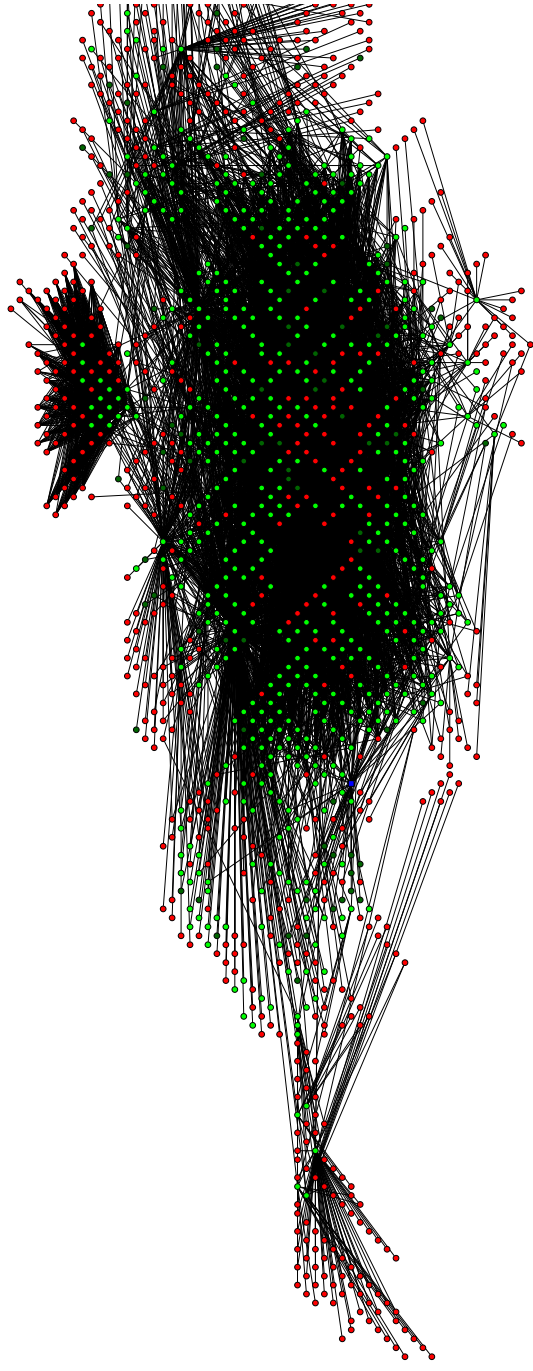


Figure 3.2: Crawl of the Ubuntu swarm. Unresponsive peers are red. Responsive peers are dark green. Responsive peers supporting PEX are green. The tracker is blue.

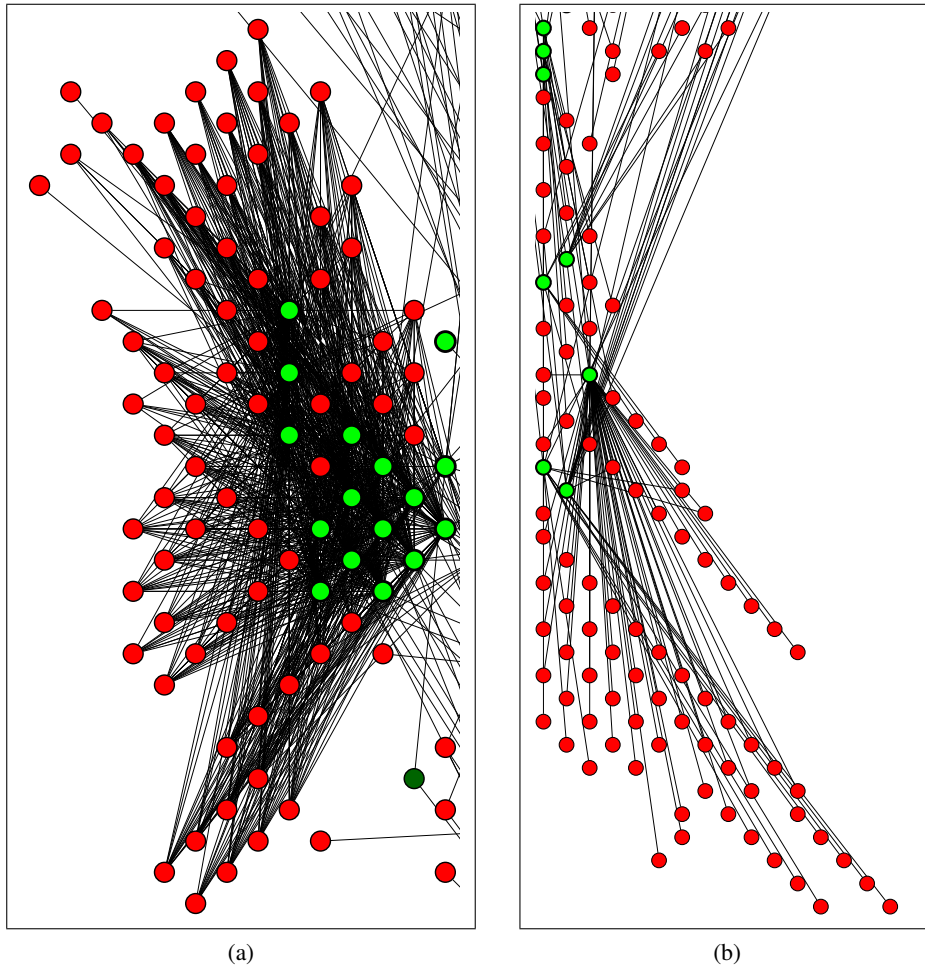
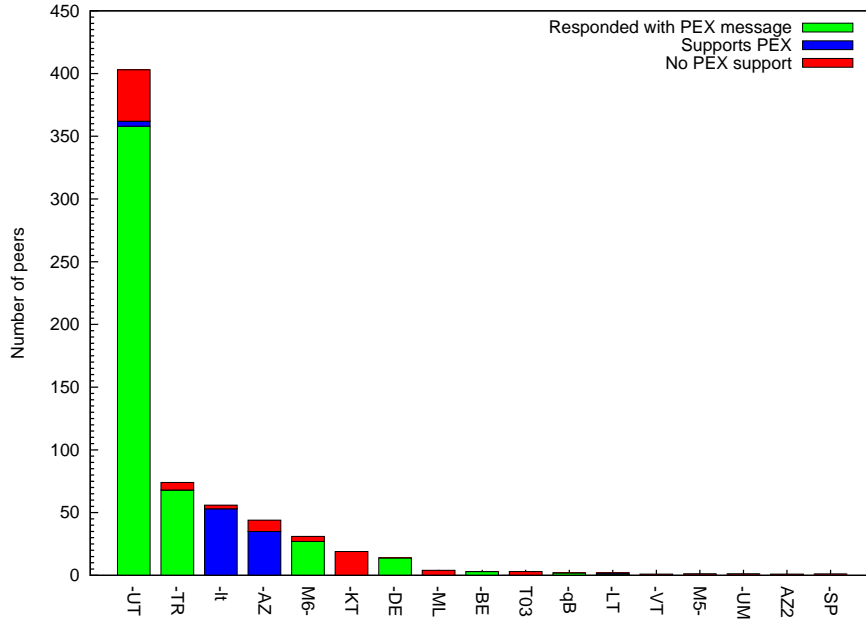
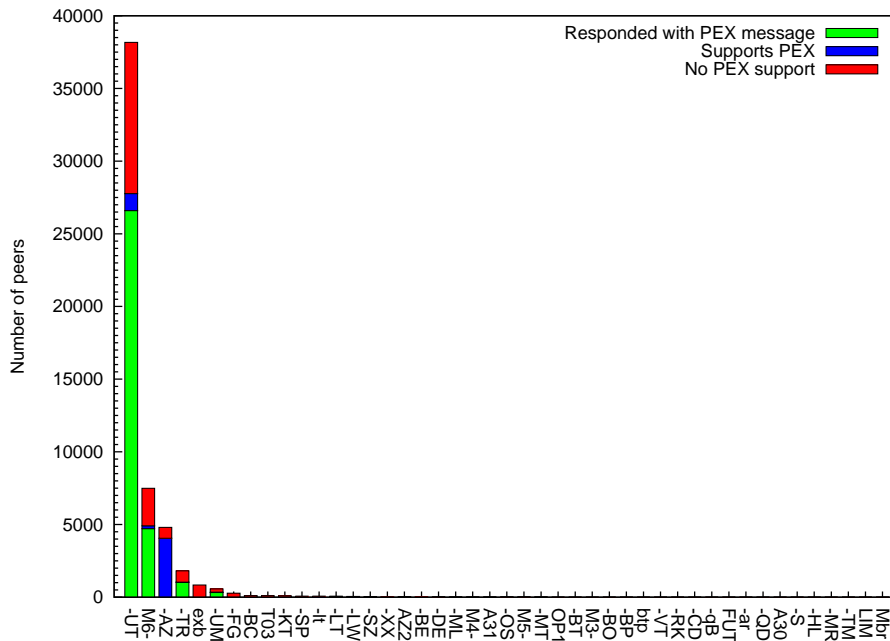


Figure 3.3: Close up of (a) the Transmission cluster and (b) the Deluge cluster. The unresponsive peers in the Transmission cluster have an IP address in the 0.x.x.x, 1.x.x.x or 2.x.x.x range. The unresponsive peers in the Deluge cluster do not belong to a specific IP range.



(a) Linux crawl (3 swarms)



(b) Public tracker crawl (10 largest swarms)

Figure 3.4: BitTorrent client versions encountered during the crawl.

See also: Table 3.1.

-AZ	Azureus (Transmission 0.7-svn)
-BE	BitTorrent SDK
-BP	Bearshare Premium P2P BitTorrent Pro
-BT	BitComet Turbo
-DE	Deluge
-HL	Halite
-LT	FDM libtorrent rtorrent ZyXEL/NSA-220
-MR	libtorrent
-MT	Movie Torrent
-OS	OneSwarm
-SP	BitSpirit
-TM	Torrent Monster
-TR	Transmission
-UM	μ Torrent Mac
-UT	μ Torrent
-VT	Vip Torrent
-XX	Transmission
-ar	aria2
-lt	libTorrent
-qB	qBittorrent (Azureus)
M5-	Mainline BitTorrent
M6-	Mainline BitTorrent
OP1	Opera

Table 3.1: Full names of BitTorrent clients encountered during the crawls.
If a short version name is not in the table, such clients did not report their full version name.

[TODO: discuss whether inserting a paragraph about connectivity matrices is useful]

3.2.2 Differences in PEX implementations and quality

PEX is not properly specified in any *BEP*². This means implementers are free to decide when and how often they send PEX messages, and how many and which peers they include in these messages. In this section we will use the data of our largest crawl, i.e. the 10 public tracker swarms. Furthermore, most subsequent graphs only include the data of the 4 most widely used, PEX supporting clients to reduce visual clutter: μ Torrent, Mainline, Transmission and μ Torrent Mac.³

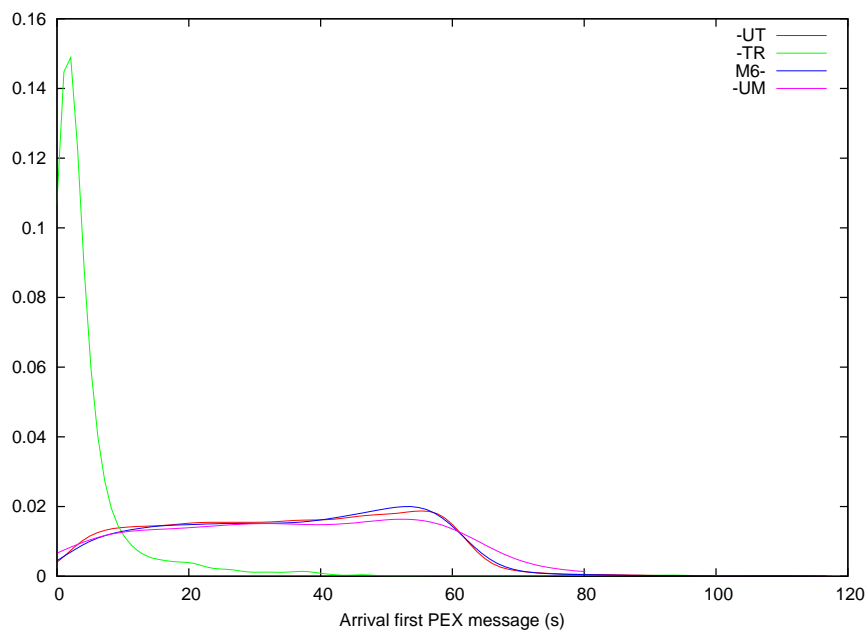


Figure 3.5: Empirical density function of the arrival of the first PEX message.

Figure 3.5 shows how soon, after the BitTorrent handshake has been completed, a client sends its first PEX message. Seemingly there are two different approaches. Transmission sends its first PEX message almost immediately after a connection has been established. The time it takes for the other clients appear to be more or less uniformly distributed. A possible explanation might be that these clients use a global timer to send periodical PEX message, and therefore the time it takes to receive the first PEX message depends on the time at which you connect to them.

When it comes to the number of peers listed in a PEX message, transmission appears to use a strict upperbound of 50 peers (Figure 3.6). The size of a PEX

²BitTorrent Enhancement Proposal, see: <http://www.bittorrent.org/beps/bep-0000.html>

³Azureus is not included as its implementation does not send PEX messages as discussed in Section 3.2.1.

message from the other clients varied much more. Part of this variation can be caused by the size of a peer's neighbourhood set. If a peer does not have many neighbours yet, it cannot send a large PEX message. These clients also seem to use a higher size limit.

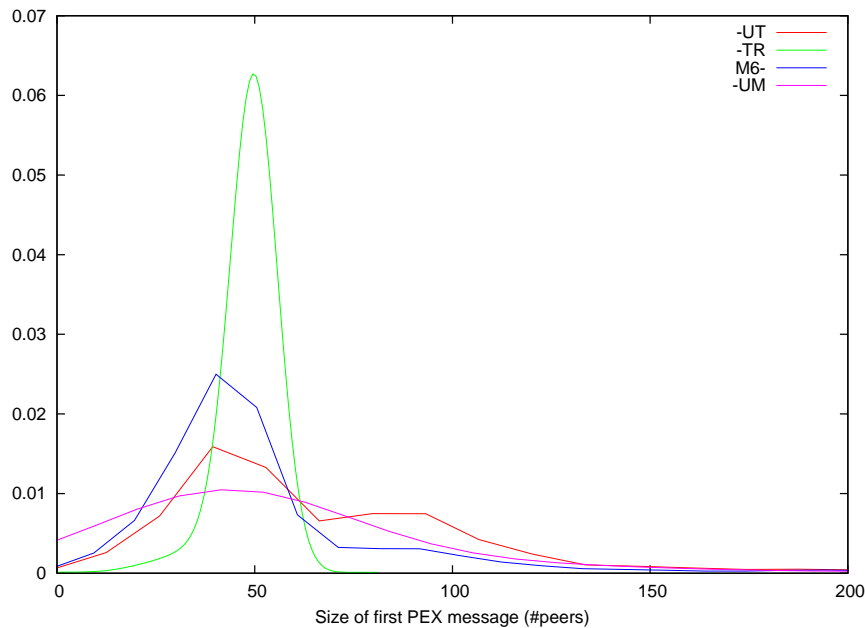


Figure 3.6: Empirical density function of the size of the first PEX message.)

It is also interesting to know how much network traffic is needed to get a single PEX message. In Figure 3.7a the size of each PEX message is plotted against the number of bytes used to communicate with the PEX message's sender. This plot shows a clear outlier: Opera (OP1) sends PEX messages containing ten thousands of peers. It is not likely that Opera sends its full neighbourhood set in its first PEX message, since not many peers would be able to maintain tens of thousands of TCP/IP connections. Perhaps it sends a list of all the peers it has seen before. If this is the case, a large portion of the peers could already be offline. Another explanation is that it may send a list of peers it has heard of. This could be a serious security flaw.

Leaving out Opera, Figure 3.7b shows that in most cases you need to spend at most 8 KB of data. The great variance in the number of bytes used for communication can be attributed to other BitTorrent messages being sent by a peer. For example, if a peer is downloading a file quite rapidly, it may send a lot of HAVE messages, informing others that it has new pieces of the file.

Remember that in Section 3.2.1 we discovered that not all discovered peers were not responsive. Since the tracker was only used initially, most unresponsive peers were reported in PEX messages. It is worth investigating whether there are a lot of

useless PEX messages and a few perfectly reliable PEX messages, or that it is the case that all PEX messages are rather poor.

Figure 3.8 shows rather unsatisfying results. A wide peak shows that on average at most 20% of the peers in a PEX message is responsive. The peak gets narrower and the percentage lower if we consider how many peers also report to support PEX, and it gets even worse if we also consider whether they actually send a PEX message.

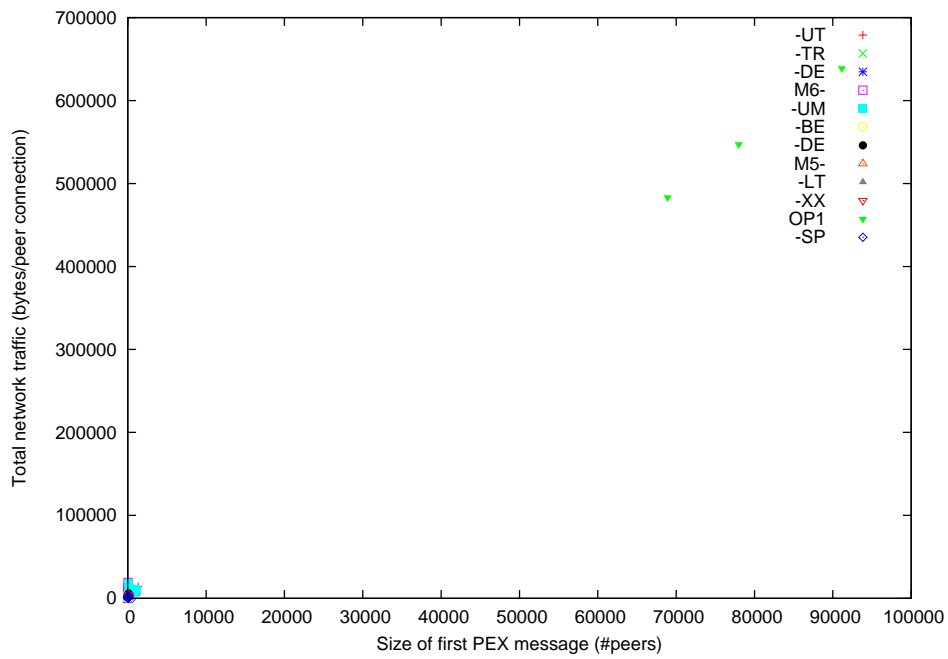
Assuming peers do not lie about their neighbourhood set, there are two reasons a peer may not respond. It either could be firewalled, or it could be congested. If a peer is congested, it is likely to have many neighbouring peers and likely to be mentioned often in other PEX messages. Figure 3.9, however, shows that the majority of unresponsive peers are mentioned less than 10 times. This might indicate most of the peers are firewalled, but also keep in mind that our crawler only collected at most 1 PEX message from each peer and could skew the distribution.

Nevertheless, BitTorrent clients could improve their PEX messages by specifying whether they are connected to the listed peers via an incoming or an outgoing connection. Peers you are connected to via an outgoing connection are more likely to be also connectible for others than peers you are connected with via an incoming connection.

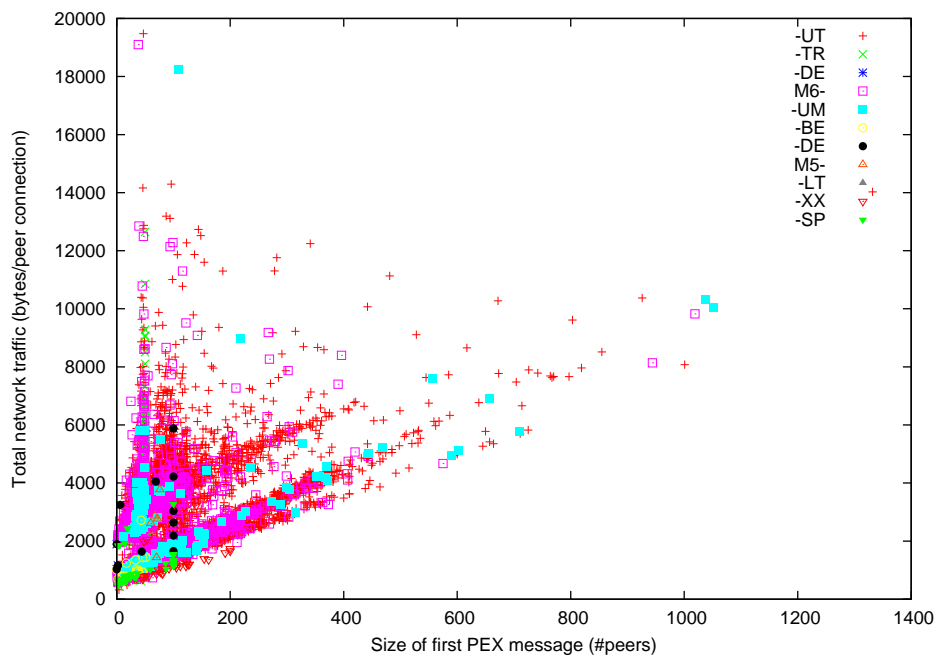
3.2.3 Remaining peer uptime

To determine how fast information received from PEX messages decays, we need to know for how long a peer stays responsive after we have last seen it. We define the remaining uptime of a peer to be the timestamp of the latest successful online check not preceded by a unsuccessful check, and the timestamp of when we closed the connection. If there is no such successful online check, we define the remaining uptime to be 0.

Figure 3.10 shows the overall remaining uptime. We can see that a quarter of the peers no longer responds after approximately 20 minutes and after 100 minutes only half remains (close up in Figure 3.12). This means that if we want to know at least n responsive peers 20 minutes in the future, we should at least now know $(1 - \frac{1}{4})^{-1} = \frac{4}{3}n$ responsive peers.



(a) All PEX sending peers from the public tracker crawl.



(b) Opera client omitted.

Figure 3.7: Number of peer stored in a PEX message versus the number of bytes received and sent to communicate with the PEX message's sender.

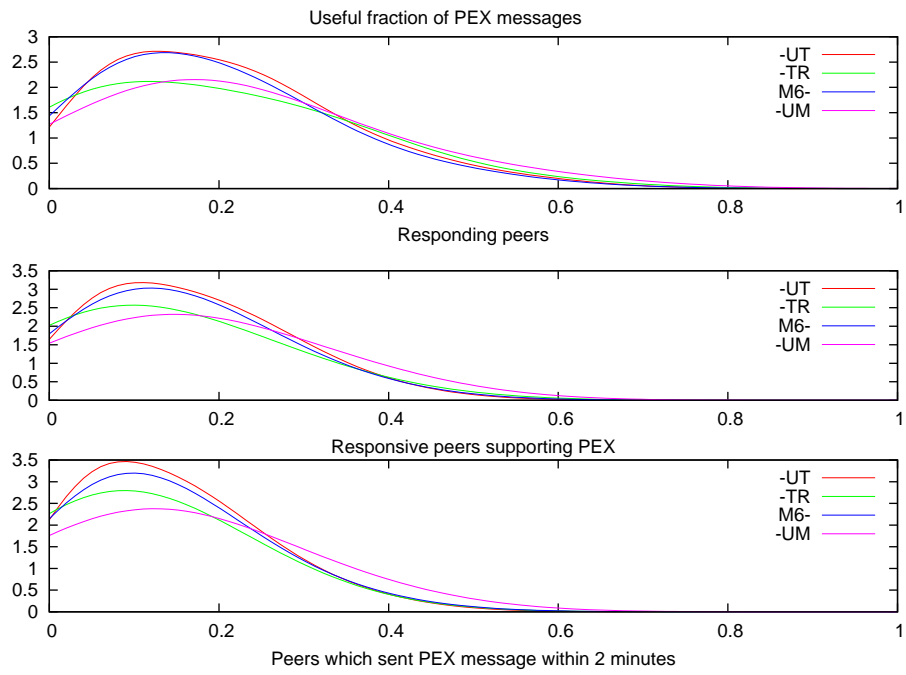


Figure 3.8: Useful fraction of a PEX message probability density.

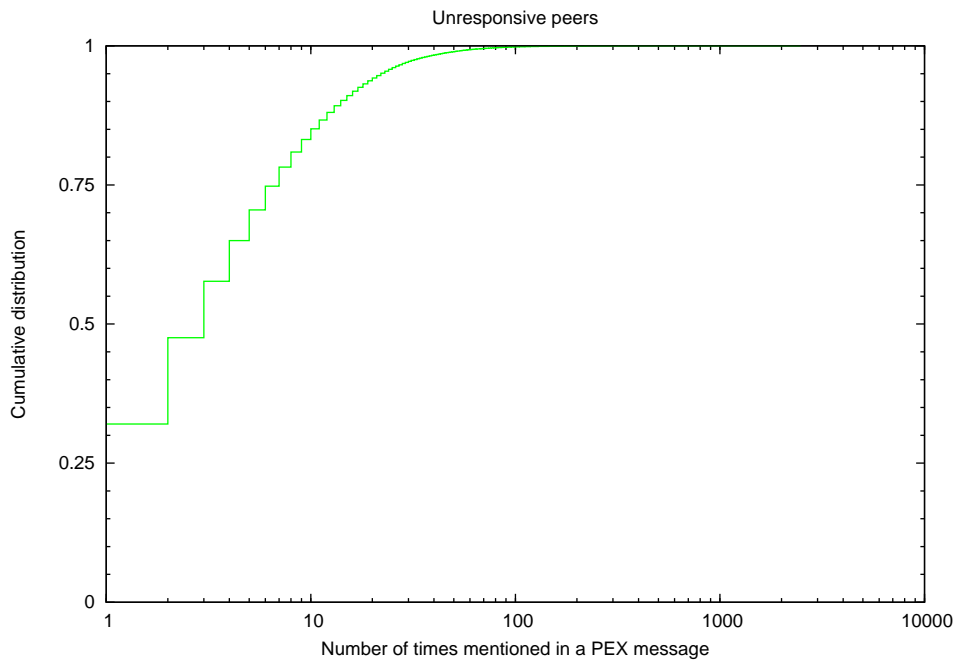


Figure 3.9: Distribution of the number of times unresponsive peers are mentioned.

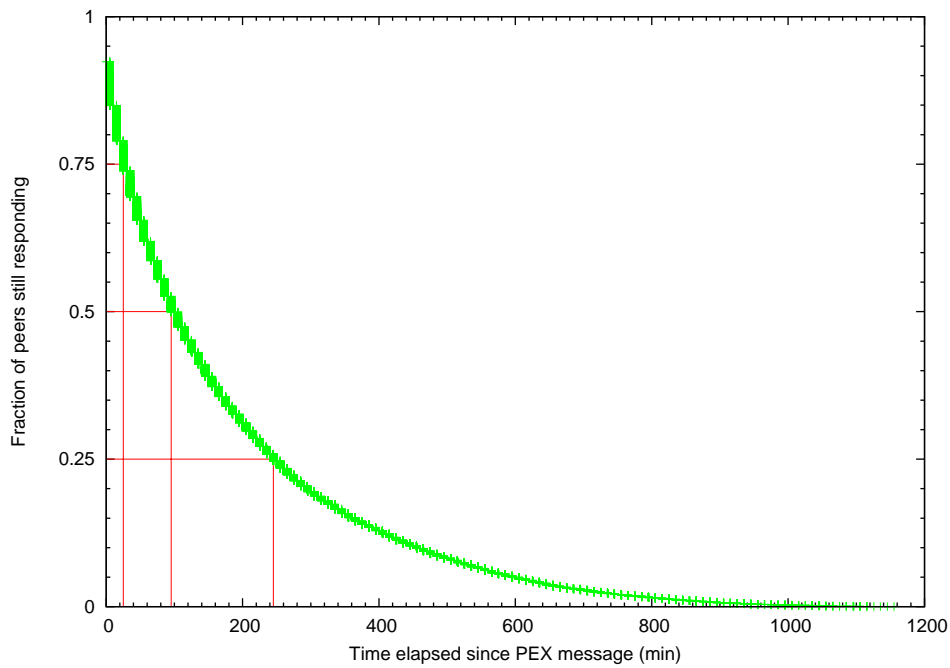


Figure 3.10: Overall remaining uptime of peers in the public tracker crawl.

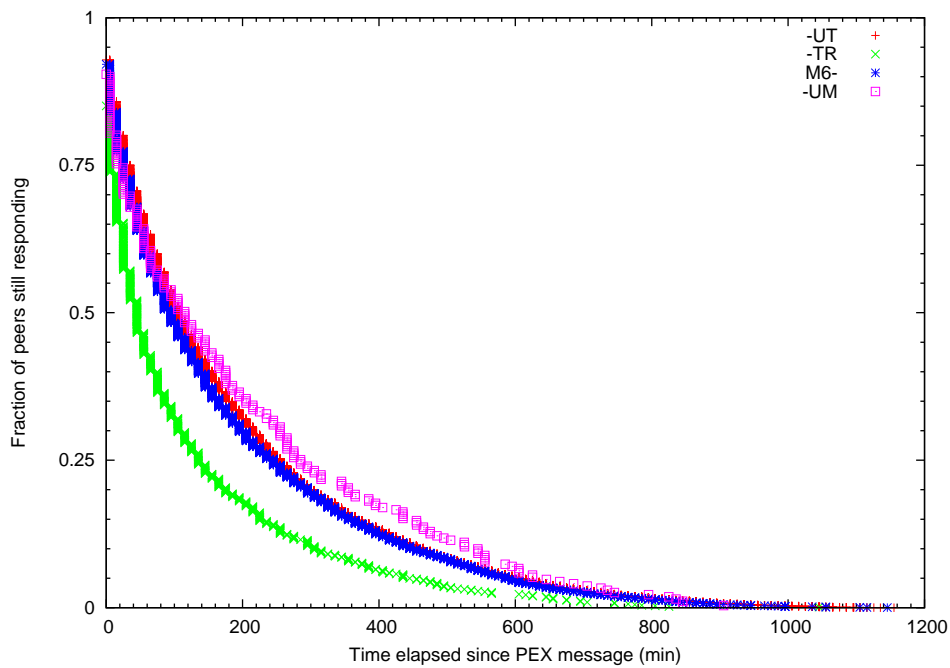


Figure 3.11: Remaining uptime of the 4 most widely used PEX supporting clients in the public tracker crawl.

[Not very useful graph...remove it?]

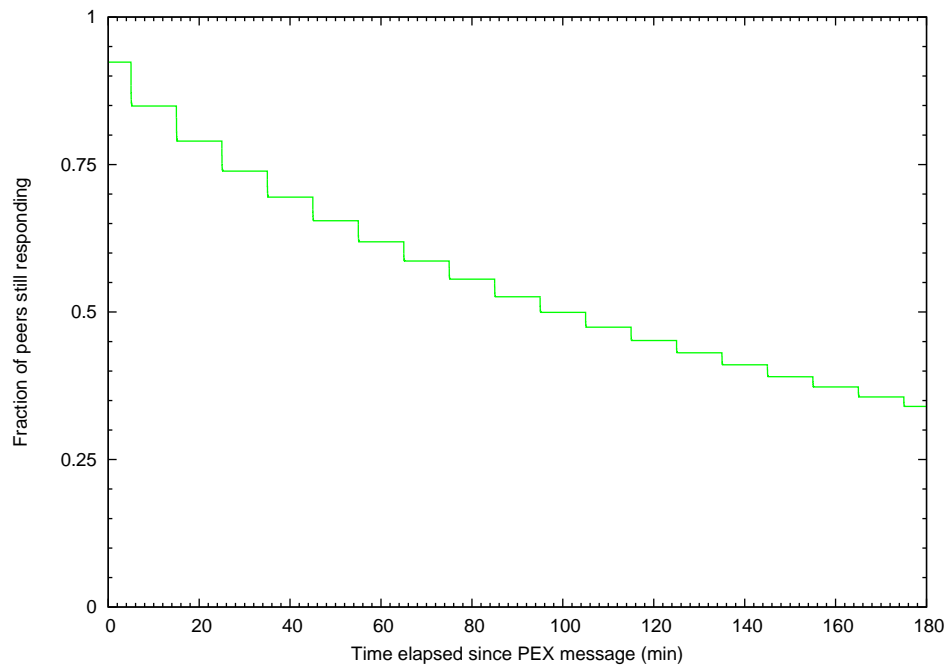


Figure 3.12: Fraction of peers still responding during the first three hours.

Chapter 4

2-Hop TorrentSmell Design

INTRO

[*Title Note:*] Should the title include the name of the algorithm or should we use something like “A Distributed Tracking Algorithm” instead (cf. chapter 4 in Jelle’s thesis, “A Decentralized Swarm Discovery Protocol”)

4.1 Some section

Perhaps precede the following two sections with a section linking together Extended-RCS and RePEX? Explain the name of 2-Hop TorrentSmell? We also have to mention something about “The latest 25 downloads” (currently not mentioned in RePEX algorithm section, since it’s only explaining RePEX itself).

4.2 Remote Content Search

[*TODO:*] Revise Extended RCS algorithm from Research Assignment report.

[*TODO: wiki:ContentSearch → 325ms latency*]

[*TODO:*] Explain the existing RCS fully here? Explain it fully in the Introduction chapter? Explain it globally in Intro chapter and more in detail here?

4.3 Tracking the swarm using RePEX

RePEX refers to the act of **re**connecting to previously encountered **PEX** peers in a swarm, with the goal of staying in touch with that swarm and potentially discover new and reliable peers. In other words, a repexing peer is a tracker.

4.3.1 Used terminology

[*Might want to move/edit this after RCS section’s been written*]

- PEX** Peer Exchange, the act of sending (deltas of) your neighborhood set to a neighbor, so the other party knows to whom you are connected [15].
- PEX peer** A BitTorrent peer that is capable of sending and receiving PEX messages.
- RePEX peer** A Tribler peer supporting the RePEX algorithm (and implicitly the related extended version of the Remote Content Search protocol).
- Repexing peer** A peer that is currently executing the RePEX algorithm for a certain swarm.
- PEX ping** Connecting to a peer and waiting for the first PEX message to arrive. A PEX ping is considered to be successful when the peer was connectable and replied with a PEX message within reasonable time.

4.3.2 RePEX algorithm

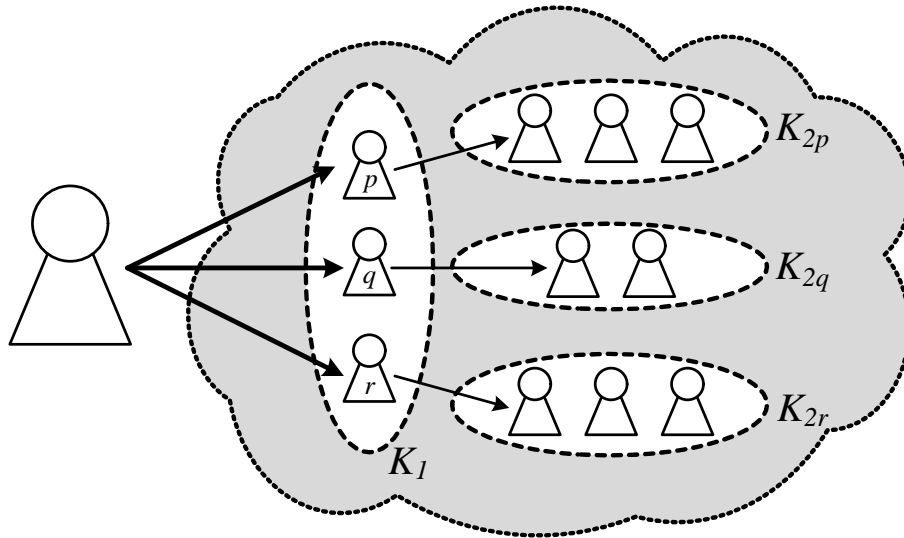


Figure 4.1: The RePEX algorithm creates and maintains a set K_1 of directly known peers and for each peer $p \in K_1$ a set K_{2p} of peers peer p directly knows.

For a given swarm, the RePEX algorithm creates and maintains a set K_1 of directly *known* peers and the sets K_{2p} of peers we know indirectly through these peers, but are directly known by peer $p \in K_1$ (Figure 4.1). These sets are restricted in *size* by configurable parameters S_1 and S_2 :

$$\begin{aligned} |K_1| &\leq S_1 \\ |K_{2p}| &\leq S_2, \quad \forall p \in K_1 \end{aligned}$$

We use K to denote the set of both directly and indirectly known peers, i.e.

$$K = K_1 \cup \left(\bigcup_{p \in K_1} K_{2p} \right).$$

To maintain the sets of peers, we will perform PEX pings. Let $A(p)$ denote the predicate whether peer p seemed to be *alive*, i.e. whether a PEX ping to peer p was successful, and let $R(p)$ denote the received PEX *reply*. For convenience, we will also use A as a filter on sets, where $A(X)$ denotes $\{x \mid x \in X \wedge A(x)\}$. To determine whether we have *checked* a peer for its aliveness, we will use C to denote the set of peers to which we have sent a PEX ping.

The algorithm then proceeds as follows:

Input: K_1
 K_{2p} for each $p \in K_1$.

Output: K'_1
 K'_{2p} for each $p \in K'_1$

1. Perform a PEX ping on each peer $p \in K_1$.
2. If $|A(K_1)| < S_1$, try PEX pinging peers $q \in (K - K_1)$ until the following condition holds:

$$|A(C)| \geq S_1 \quad \vee \quad C = K$$

Verbally, continue to PEX ping until we have found (at least) S_1 peers that are alive or until we have checked all known peers.

3. If we have not found enough peers, i.e. $C = K$ and $|A(K)| < S_1$, use a bootstrap mechanism, e.g. some form of distributed tracking or a central tracker, to get a set of peers B and PEX ping those until the following condition holds:

$$|A(C)| \geq S_1 \quad \vee \quad C = K \cup B$$

4. For some \hat{C} , construct a new set $K'_1 = A(K_1) \cup \hat{C}$, where

$$\begin{aligned} \hat{C} &\subseteq A(C - K_1), \\ |\hat{C}| &= \min(S_1 - |A(K_1)|, |A(C)|). \end{aligned}$$

In other words, construct a new set of connectable and pexing peers using all alive peers, preferring peers from K_1 .

5. Similarly, for each $p \in K'_1$ and for some \hat{R}_p , construct a new set $K'_{2p} = \hat{R}_p$, where

$$\begin{aligned} \hat{R}_p &\subseteq R(p), \\ |\hat{R}_p| &= \min(S_2, |R(p)|). \end{aligned}$$

Implementation notes

[*Move to next chapter?*] In the described algorithm framework implementation details are deliberately omitted. It is up to the implementor to decide e.g. whether peers are pinged in parallel, whether sophisticated selection criteria are used to construct the final peer sets, and whether steps of the algorithm are executed in a pipelined fashion. The framework also does not specify how often the algorithm has to be executed.

4.3.3 Current Implementation

[*Move to next chapter?*] In this section we describe the current implementation of the described algorithm. We will cover the used data structure (Section [*RefTODO*]), PEX filtering (Section [*RefTODO*]) and the current parameters (Section 4.3.3).

Data Structure

The current implementation stores previously encountered PEX peers in a so called *SwarmCache*. The *SwarmCache* is a Python dictionary (depicted in Figure 4.2) representing the set K_1 from the algorithm in Section 4.3.2. It associates the (IP,port) address of a peer p in K_1 with another dictionary containing the last-seen timestamp and a list of peers corresponding with set K_{2p} . Optionally, this dictionary contains a flag indicating whether this peer was alive in the previous iteration of the algorithm.

Key	Value	
(IP,port)	Key	Value
	'last_seen'	Timestamp when this peer was last seen alive.
	'pex'	[((IP,port), PEX flags)], representing K_{2p} .
	'prev'	Optional flag value, indicating this peer was also alive in the previous iteration of the algorithm.

Figure 4.2: The *SwarmCache* datastructure.

We store the last-seen timestamp of each peer to compute the *age* of a *SwarmCache*. Currently we define the age to be the difference between now and the largest timestamp found in the *SwarmCache*. When a *SwarmCache* becomes too old, we will update it by executing the RePEX algorithm.

The list of peers stored under the 'pex' key also contains so called PEX flags. The PEX flags are stored in a single byte as described in [15], but are not used in the current implementation. Future versions can use these flags in deciding how to construct the K'_{2p} from Section [*RefTODO*].

The 'prev' flag is currently only used for measurement purposes.

Implementation Parameters

The current implementation is configurable by changing the configuration parameters. The parameters themselves are currently module variables, but should probably be moved to the Session config in the future. Below we describe each parameter and its current (arbitrarily chosen) value.

`REPEX_SWARMCACHE_SIZE = 4`

The preferred size of the SwarmCache. This parameter corresponds with S_1 in Section 4.3.2.

`REPEX_STORED_PEX_SIZE = 5`

The number of peers to sample from a PEX message. This parameter corresponds with S_2 in Section 4.3.2. The sampling takes place right after receiving and filtering the PEX message.

`REPEX_PEX_MINSIZE = 1`

This parameter acts as a filter. If a received PEX message is too small, we do not consider the PEX ping to be successful. Note that this parameter is dangerous for small swarms.

`REPEX_INTERVAL = 20*60`

How often a SwarmCache needs to be refreshed in seconds. The algorithm is executed for a SwarmCache if its age is larger than this parameter's value.

`REPEX_MIN_INTERVAL = 5*60`

The minimum number of seconds between RePEX attempts. In case running the RePEX algorithm fails for a certain swarm and produces an empty SwarmCache, we do not want to start another attempt immediately. Doing so might result in starvation.

`REPEX_PEX_MSG_MAX_PEERS = 200`

PEX messages containing more peers than this parameter are truncated.

`REPEX_LISTEN_TIME = 50`

The number of seconds within a PEX message must arrive after a successful BitTorrent connection is made for a PEX ping. If a PEX message did not arrive in time, the PEX ping is considered not to be successful.

`REPEX_INITIAL_SOCKETS = 4`

The maximum number of sockets used initially when executing the RePEX algorithm.

`REPEX_MAX_SOCKETS = 8`

The maximum number of sockets used when executing the RePEX algorithm after a certain condition has been met: Either a PEX ping failed for a peer in K_1 or all peers in K_1 have been checked.

`REPEX_SCAN_INTERVAL = 1*60`

How often the RePEX scheduler scans for outdated SwarmCaches in seconds.

Chapter 5

Implementation and Experiments

Intro

[Merge with deployment chapter?]

5.1 Cost of RePEX

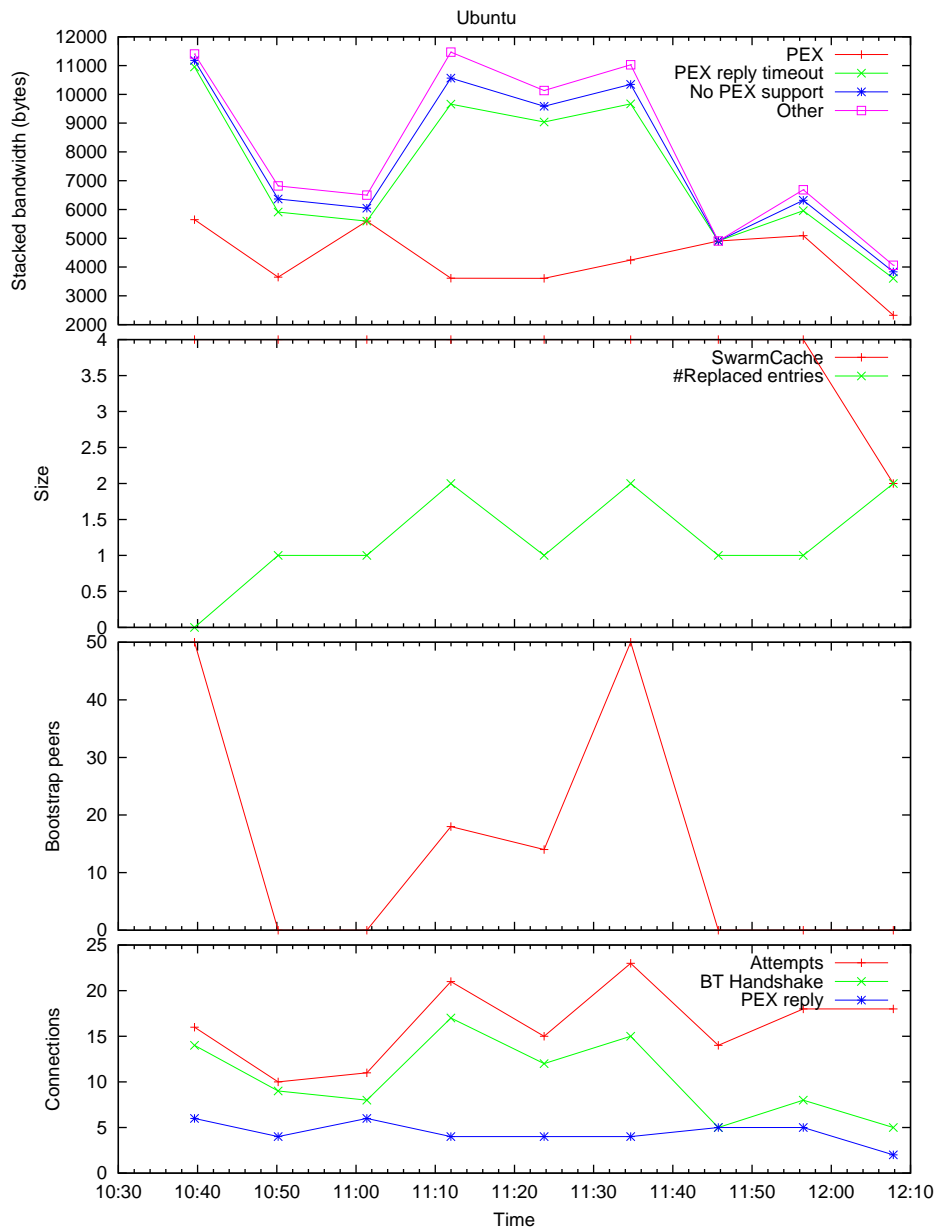


Figure 5.1: This figure needs some readjusting. Note that last data point might be faulty because of network connectivity issues

Chapter 6

Deployment

(This chapter has to be written after deployment measurement)

6.1 Foo

Foo

[*include:*]

[*Current architecture*]

[*Rerequester cannot see origin of peers*]

[*All found peers are given to Encrypter (etc.)*]

[*Connections are made to all IPs, unless the Connector has reached max. conns*]

Chapter 7

Conclusions and Future Work

7.1 Conclusions

TODO CONCLUSIONS

7.2 Future Work

TODO FUTURE WORK

Bibliography

- [1] ANT Censuses of the Internet Address Space.
<http://www.isi.edu/ant/address/>.
- [2] AzureusWiki: Distributed hash table.
http://www.azureuswiki.com/index.php/Distributed_hash_table.
- [3] BitTorrent.org: DHT Protocol, 2008.
http://www.bittorrent.org/beps/bep_0005.html.
- [4] 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.
- [5] B. Cohen. Personal announcement: BitTorrent - a new P2P app.
<http://finance.groups.yahoo.com/group/decentralization/message/3160>.
- [6] B. Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, volume 6. Berkeley, CA, USA, 2003.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] P2P-Next. <http://www.p2p-next.org/>.
- [11] 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.
- [12] J. Roozenburg. Secure Decentralized Swarm Discovery in Tribler. MSc thesis, Delft University of Technology, November 2006.
- [13] 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.
- [14] 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.

- [15] Transmission: Extended messaging.
<http://trac.transmissionbt.com/browser/trunk/doc/extended-messaging.txt>.
- [16] Tribler – BitTorrent client. <http://www.tribler.org>.
- [17] Tribler Wiki: 4th Generation of P2P.
<https://www.tribler.org/trac/wiki/4thGenerationP2P>.