Swarm Discovery in Tribler using 2-Hop TorrentSmell

Raynor Vliegendhart

Delft University of Technology

Swarm Discovery in Tribler using 2-Hop TorrentSmell

Master's Thesis in Computer Science

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

Raynor Vliegendhart

31st May 2010

Author
Raynor Vliegendhart

Title
Swarm Discovery in Tribler using 2-Hop TorrentSmell

MSc presentation
16th June 2010

Graduation Committee

prof. dr. ir. H. J. Sips (chair)	Delft University of Technology
dr. ir. J. A. Pouwelse	Delft University of Technology
dr. M. M. de Weerdt	Delft University of Technology

Abstract

Peer-to-peer (P2P) technology allows us to create self-organising and scalable systems. The distributed nature of these systems requires a solution for finding interesting peers. In the context of P2P file sharing, finding peers who are downloading the same file is referred to as the *swarm discovery* problem. In BitTorrent, this problem is solved using a central server, called a tracker. This central component hinders the scalability of BitTorrent.

We have designed a distributed swarm discovery algorithm, called 2-Hop TorrentSmell. It is composed of two parts. The first part builds on top of an existing keyword search system to find peers who have recently downloaded a certain file. The second part consists of an algorithm called RePEX, which allows a peer to stay in touch with swarms it is no longer in by periodically recontacting previously encountered swarm members. Our RePEX algorithm leverages a widely used BitTorrent extension for swarm discovery called PEX. We have conducted a study to understand the reliability and usability of PEX to optimize the design of our solution. We have implemented the RePEX part of 2-Hop TorrentSmell as an addition to the Tribler P2P network.

For the evaluation of our RePEX algorithm, we have tested it on the 10 largest swarms on a public tracker. In addition, we have deployed the algorithm in a beta version of Tribler and let it run on swarms the user has downloaded from. Evaluation results show our algorithm is scalable and effective in popular swarms.

Preface

This is the report of my MSc thesis project in Computer Science on the subject of distributed swarm discovery for the Tribler peer-to-peer network. The research has been performed at the Parallel and Distributed Systems Group of the Faculty of Electrical Engineering, Mathematics, and Computer Science of Delft University of Technology in the context of the I-Share research project.

I would like to thank many people who have helped me during this project. First of all, I would like to thank my supervisor Johan Pouwelse for his overall guidance and support. My thanks also go to Arno Bakker and Boudewijn Schoon for their great help with getting familiar with the architecture of Tribler and their feedback on my proposed implementations. Thanks to Daniel Peebles for his suggestion on exponentiating adjacency matrices. Thanks to Jip Man Vuong for his feedback on some of the figures in this thesis. I am grateful to my proofreaders Luite Stegeman and Gertjan Halkes and their invaluable feedback. Finally, I would like to thank prof. dr. ir. H. J. Sips for chairing the examination committee and M. M. de Weerdt for participating in the examination committee.

Raynor Vliegendhart

Delft, The Netherlands
31st May 2010

Contents

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. P2P has come a long way and is now even used for video applications [27, 33, 43].

Unfortunately, building P2P applications to operate in a fully distributed way is not easy. The past has shown that designers often choose the easy way out and sacrifice fully distributed operation by introducing central components in their systems. These central components prevent a P2P system to become fully self-organising 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 action against Napster, where it was forced to shut down its centralized search facility [24].

Removing the need for any central component or server is a key aspect of the 4th Generation of P2P [39] 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 [38] and is part of the P2P-Next project funded by the European Union [29]. The goal of of the P2P-Next project is to build a new platform for P2P television. In this thesis, we will design an initial version of a fast and scalable distributed solution for finding peers in a swarm for the Tribler client, that will eventually replace the central tracker component of the underlying BitTorrent protocol. We will opt for an incremental approach, where we can evaluate the simple initial solution and detect possible deficiencies early on.

The remainder of this introductory chapter gives background information on file sharing in BitTorrent. The BitTorrent protocol is described in Section 1.1. In Section 1.2 we describe the socially enhanced Tribler BitTorrent client and its relevant features for this thesis. In Section 1.3 we list our contributions, and finally, in Section 1.4 we outline the remainder of our thesis.

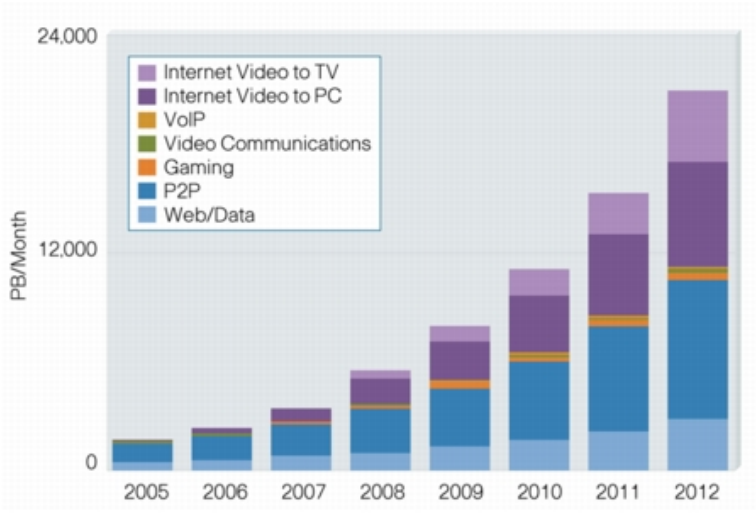


Figure 1.1: Global consumer Internet traffic forecast. Source: Cisco, 2008 [11].

1.1 BitTorrent

BitTorrent is a file sharing P2P protocol created by Bram Cohen in 2001 [12]. 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 [13].

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* [13]. 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. An additional way to find new peers is through the Peer EXchange (PEX) extension, which allows peers to share lists of swarm members with each other.

1.2 Tribler

Tribler is a BitTorrent client originally based on the open source ABC client [1], that is enhanced with social features. Tribler adds, amongst others, the notions of identity, friends and people with similar tastes to BitTorrent. The basic ingredient for all these social features is memory. Unlike other BitTorrent clients, Tribler uses a database to carry knowledge from one session to another.

In Tribler, each peer has a secure *permanent identifier*, or *PermID* for short [41].

This identifier can be used to create groups of friends and associate users with similar download tastes. This identity is needed to create a social network, which can provide additional services, e.g. cooperative downloading [21].

Tribler peers can discover each other and content using the epidemic gossip protocol called *BuddyCast* [31]. Periodically, a Tribler peer chooses another peer to exchange a BuddyCast message with. A BuddyCast message contains a list of preferences describing the Tribler peer's 50 latest downloads, a list of Tribler peers with similar tastes and a list of random Tribler peers. Tribler peers tend to connect to users with similar download tastes to form a semantic overlay. A BuddyCast message also contains a list of discovered torrents, which the peer may have injected itself into the Tribler network or may have heard of from other Tribler peers.

Information received via BuddyCast messages are stored in a database, called the *MegaCache* [30]. This cache serves as the peer's memory. A recent feature added to Tribler is the ability to perform a *keyword search* that not only searches the peer's own memory, but also the MegaCaches of neighbouring Tribler peers [40]. This allows us to find content that we ourselves have not heard of.

1.3 Research question and contributions

The subject of this thesis is designing a fully distributed solution to the swarm discovery problem, i.e. finding peers in a swarm. In this thesis, we make the following main contributions:

- We study the behaviour of different implementations of the widely used PEX BitTorrent extension. This extension solves a large part of the problem central to this thesis.
- We present the design of our distributed swarm discovery algorithm called 2-Hop TorrentSmell, relying on existing Tribler features and the widely used PEX protocol.
- We implement the part of 2-Hop TorrentSmell called RePEX, that is responsible for tracking peers and we evaluate its performance.

These contributions are a first step towards a fast and fully distributed swarm discovery solution. The goal is to eventually replace the central tracker in BitTorrent altogether.

1.4 Thesis outline

The remainder of this thesis is organized as follows. We describe the swarm discovery problem and the currently deployed solutions and their drawbacks in Chapter 2. In Chapter 3 we study the behaviour of the PEX protocol in order to understand its effectiveness and deficiencies. The knowledge gained from this study is used in the two following chapters. In Chapter 4 we present the design of our 2-Hop TorrentSmell solution to the swarm discovery problem. We have implemented a part of the 2-Hop TorrentSmell called the RePEX tracking algorithm, which we evaluate in Chapter 5. Our conclusions and proposed future work is presented in Chapter 6.

Chapter 2

Problem Definition

This chapter defines the main subject of this thesis: swarm discovery in a scalable and effective way. Swarm discovery is the problem of finding peers that are downloading the same file, which we describe in Section 2.1. We define the requirements for a solution in Section 2.2. In Section 2.3 we discuss that none of the currently deployed solutions meet all the requirements.

2.1 Swarm discovery

In traditional client-server architectures, all content is stored at the server, but in P2P networks, content is distributed over all peers. Since there is no central location in a P2P network to fetch content, a search mechanism must be in place to find the peers that have the particular content the user is interested in. The dynamics of a P2P network can complicate the problem of finding these peers.

In work related to this thesis, Roozenburg defines *swarm discovery* to be the problem of finding peers in a specific *download swarm* [32]. In BitTorrent, a swarm consists of peers downloading the same content. These peers are interested in locating each other, to allow them to barter file pieces.

In order to barter with a peer, we need its public IP address and listening port. Hence, this information about swarm peers needs to be found when solving the swarm discovery problem. We will refer to a list of these network addresses of peers in a certain swarm as a *peer list*.

2.2 Design requirements

A quality solution to the swarm discovery problem should adhere to the requirements listed below. We will use these requirements to review currently deployed solutions and to guide the design of our solution presented in Chapter 4.

Bootstrapping

Bootstrapping is a special startup phase 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 searches for a small number of peers that are in the swarm it wants to join. This search involves contacting known peers that are not in the swarm, but possibly know who are in it or know other peers that possibly know this. The initial swarm peers found via this search can then be used instead for further swarm discovery. If these initial peers cannot be found, further swarm discovery becomes impossible.

Note that when a centralized solution like a central tracker is used, bootstrapping is not necessary. The address of the tracker is statically known and thus both initial and subsequent peers can always be found using the tracker. In a decentralized setting, however, bootstrapping is always a required and important step.

Scalability

It is desirable that a swarm discovery solution is lightweight and scales well. The overhead required to solve the swarm discovery problem should be minimal in order not to hinder the actual downloading of content. This overhead consists of CPU usage, storage space and bandwidth. The latter resource is the most scarce and is very valuable in the BitTorrent protocol for bartering purposes. The costs of running a swarm discovery algorithm should grow slowly with the number of simultaneous downloads and swarm sizes.

Speed

The speed of the swarm discovery solution should be high, allowing peers to find others quickly and start downloading right away. This is especially desirable in applications like P2P video streams [27] where the user wants to watch videos without much delay. Needless to say, the speed of the solution should scale well as well.

Security

Security is an important aspect of any distributed solutions. Neglecting security issues could result in an ineffective, or even worse, exploitable P2P network. One of the possible attacks on a P2P network is to pollute or poison its indexes, as described by Liang et al. in [22]. Projecting this type of attack onto the swarm discovery problem, it basically means that an attacker is spreading false 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 wreak havoc on the victim. To avoid such attacks, an attacker must be limited in its ability to spread false information. One possible, but simple solution is having peers validate information the moment they receive it. It prevents false information to be propagated, but can itself be exploited to send unsollicited data packets to a victim [35].

Effectiveness

Peers discovered through a swarm discovery algorithm should be reachable, otherwise the algorithm is not effective. To be effective, a good swarm discovery solution takes connectivity issues and churn into account.

In a perfect system, all peers in a peer list are connectable. In reality, however, firewalls and NAT routers can reduce the connectivity of a peer. To the outside world, that peer may not appear to be connectable. Hence, it would be fruitless to include such a peer in a peer list.

When connectivity issues are not taken into account, any algorithm to find peers becomes less effective, as the following anecdote illustrates. An early version of Tribler's epidemic gossip protocol called BuddyCast did not check whether a peer was connectable. As a result, nearly 60% of the messages were unreliable, containing peer addresses of which over 80% were either offline or unconnectable [31]. A simple mechanism that was added to a later version allowed 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 more reliable BuddyCast messages.

While connectivity issues cause peers not be reachable to begin with, churn causes peers to no longer appear reachable while they once were. Churn is the dynamics of peer participation in a P2P network, i.e. the independent arrivals and departures of peers. It 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.

2.3 Existing solutions

No solution exists which fully meets all the requirements. The currently deployed solutions are:

[1] For example, Azureus' DHT consists of more than 1 million nodes [18].

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

The first version of the BitTorrent protocol used a single central tracker to solve the swarm discovery problem. Support for multiple trackers was added later [8]. To decentralize BitTorrent, two distributed solutions based on a Distributed Hash Table were independently developed [5, 9]. Two different Peer Exchange protocols were developed to minimize the load on the trackers and the DHT [6, 28]. Both the two DHTs and the two PEX protocols are incompatible.

We will now describe the three deployed solutions in more detail and discuss why they do not fulfill all mentioned requirements. For PEX, we will only focus on the widely supported ut_pex implementation [28, 37] developed by the µTorrent developers [42] in this thesis.

Central trackers

Central trackers have been used since the first version of BitTorrent. The role of a tracker is to maintain peer lists for swarms. Peers contact a tracker that is known to track the swarm they are interested in to discover other peers in that swarm, but also to announce their own presence [13]. Communication with the tracker is done using HTTP GET requests. When a peer has not announced its own presence for a while, the tracker removes that peer from the peer list. An example scenario of a new peer joining a swarm is shown in Figure 2.1.

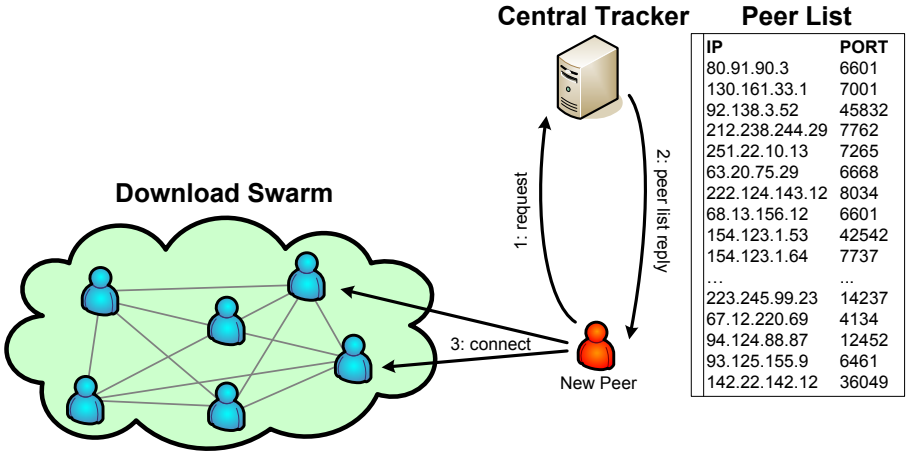


Figure 2.1: A new peer joining a swarm using a central tracker.

The peers know which tracker to contact since its address is stored in the torrent file, which also contains other swarm metadata like the file name and integrity hashes. As mentioned in the previous section, a centralized solution obviously does not need a bootstrap step.

However, a centralized solution is not scalable. While the cost of servicing a single peer list request is low, the bandwidth costs of running a tracker grow linearly with the number of peers. A measurement in [32] shows that a tracker under a high load can take a few minutes to respond, or may not even respond at all half the time.

From a security point of view, the central tracker listed in a swarm's metadata file is trusted by the peers. They have no reason to believe the tracker will behave maliciously. The tracker is, however, susceptible to attacks. An attacker can announce its presence for a targeted swarm while not running the BitTorrent protocol in order to poison the tracker's peer list. Peers that discover the attacker via the tracker will waste time and bandwidth trying to connect to the attacker's address. For this poison attack to be effective, the attacker must perform many announces using different source ports and optionally from multiple hosts with different IP addresses. Since peers attempt to contact the attacker, this attack requires sufficient bandwidth.

Certain trackers that allow a peer to specify a public IP address in an announce are more susceptible [36]. Attackers can abuse this feature and send announces with fake IP addresses using a single machine. This variation of the poisoning attack is cheap as it does not require much downstream bandwidth, unlike the previously described attack. The ability to specify a public IP address also allows attackers to announce the address of a targeted host that is not in the swarm. If sufficient peers in the swarm discover this victim's address, the victim's host will be overwhelmed by connection attempts.

Churn is handled by trackers by having peers announce themselves periodically. Peers that have not announced their presence recently are removed from the peer list. While it is not listed in the BitTorrent protocol specification [10], peers usually send an announce every 30 minutes [14]. The default timeout used by the original BitTorrent tracker implementation is 45 minutes [7]. This means that information received from a tracker can be up to 45 minutes old.

While not specified in the BitTorrent protocol, there are trackers –e.g. Mainline's tracker implementation [7]– that take connectivity issues into account and perform a check whether an announcing peer is connectible. Such checks increase the load of the tracker, but have the benefits of having a higher quality peer list and being less susceptible to poisoning attacks.

Distributed Hash Table

Instead of using a central server to store a swarm's peer list, the peer list for a swarm can also be stored in a Distributed Hash Table. Like normal hash tables, a DHT stores key-value pairs. To avoid confusion, we will use "node" to mean a host that is part of the DHT and "peer" to mean a BitTorrent client in a swarm. Each node is responsible for storing the values for the keys that are closest to its own node identifier. In the BitTorrent DHT, a peer list is stored under the corresponding swarm's *infohash*, i.e. the swarm identifier stored in the swarm's torrent file.

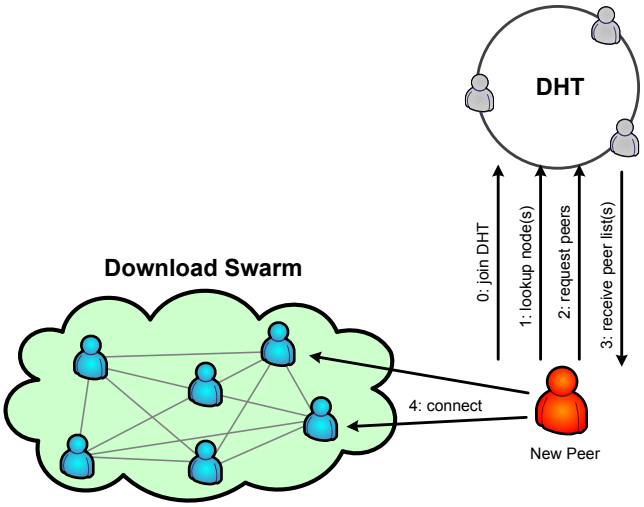


Figure 2.2: A new peer joining a swarm using a DHT.

Assuming a peer has already joined the DHT, it can solve the swarm discovery problem for a certain swarm as follows. First it uses the swarm's infohash as a key to look up. The lookup message will be routed through the DHT until it hits one or multiple nodes responsible for this key. The peer can then query these nodes for a peer list for the particular swarm it is interested in. Afterwards it will also announces its own presence to these nodes, such that they can update their peer lists. If the peer wants to be discoverable via the DHT by other peers, it should periodically lookup the responsible nodes and announce its presence. An example scenario of a new peer joining a swarm using the DHT is shown in Figure 2.2.

The hard part in this whole process is to actually join the DHT. For this, a peer has to bootstrap on an existing node in the DHT. This node can supply the peer with the required information to build a routing table needed to join the DHT network and operate normally. The way BitTorrent clients find an initial DHT node varies. The Mainline client only uses the DHT for swarms without a central tracker and finds the inital nodes' addresses in a special field of the corresponding torrent file [9]. Azureus and Mainline-compatible clients use peers encountered in swarms as initial DHT nodes [14]. As a backup, some clients use a list of hardcoded known DHT nodes.

An apparent advantage of a DHT is the ability to share the load over many nodes, making it scalable. It is, however, possible that a node with limited resources is responsible for a fairly popular key. To mitigate this problem, peer lists are replicated among the k closest nodes. The value of k is 3 for the Mainline DHT and 20 for the Azureus DHT. The Azureus DHT also has additional means for dealing with sudden load increases by migrating stored values for a popular key to 10 alternate locations when needed [14].

Unfortunately, this also illustrates a weakness of the DHT. A DHT node is only

related to a swarm by its own node identifier being close to the swarm's infohash. Although possible, it is generally not the case that it is a downloader in that swarm. Since it has no semantic relation with the swarm, it can only observe its own load and react to it. Load balancing would be easier if there was a semantic relation, as it would mean that popular swarms would automatically give rise to more nodes tracking them.

The rigid structure of DHTs also poses some security issues. 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 [35], 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:* 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.

These amplifications techniques can be used in the BitTorrent DHTs. For example, Mainline 5.2.2 unconditionally includes the sender of a DHT ping message in its routing table [7].

To make DHTs more resistant, it is necessary that an attacker's ability to 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 multifake technique can be rendered less effective by bounding the communication from a node to a single physical address, but this may give rise to a *disconnection attack* [35]. It is also hard to prevent redirection attacks. It is important to validate whether a node C is actually 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 [17]. Luckily, the index poisoning attack described in [22] is ineffective. The BitTorrent DHTs only allow peers to announce themselves and thus can only store their own IP under a key.

Since the two deployed DHTs do not do any checking, they also do not check whether a node is connectable. A study by Crosby and Wallach [14] 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 unable to reach them.

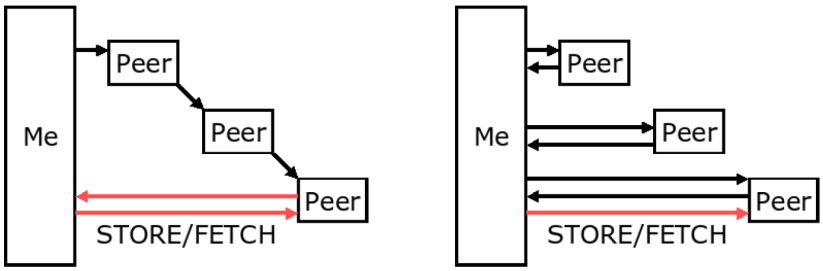


Figure 2.3: (a) Recursive versus (b) Iterative routing [14].

To cope with churn, the Kademlia DHT algorithm on which the two BitTorrent DHTs requires nodes to periodically republish keys by design [25]. In the BitTorrent DHTs, this means peers are required to periodically announce that they are still active in certain swarms if they do not want to be removed from the stored peer lists.

How messages are routed in a 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 and receiving all messages (Figure 2.3a versus 2.3b). In the two BitTorrent DHTs, which both use iterative routing, 20% of the nodes contacted in a single lookup are no longer alive in 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. As a result, lookups can take over 1 minute to complete [14]. If the BitTorrent DHTs were to use a recursive routing scheme, each node along the lookup path would have the opportunity to discover dead nodes in its routing tables.

Peer Exchange

Peer EXchange (PEX) is a BitTorrent extension designed to speed up swarm discovery. Instead of relying on peers supplied by a central tracker or the DHT, peers can share their own neighbourhood set with their neighbours, as shown graphically in Figure 2.4. After a peer has exchanged lists with another peer, it may connect to the newly discovered peers.

In more detail, PEXing is done as follows. For each PEX-capable link, i.e. a link for which both endpoints support PEX, a peer maintains a set of peer addresses it has already sent to the other party. When a peer decides to send a new PEX message, it sends the difference between its current neighbourhood set and its set of peers already sent, or a subset hereof if the resulting set is too large. Computed using these same two sets, the same PEX message also contains a set of previously sent peers which the peer is no longer connected to since the last PEX message.

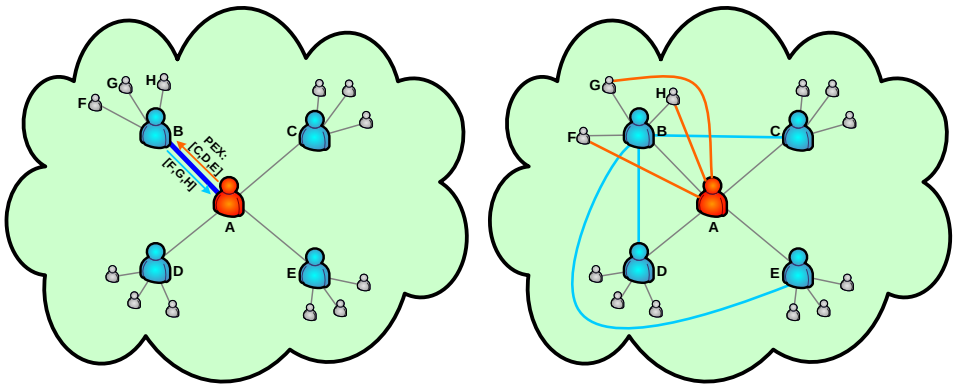


Figure 2.4: (a) Peer A and peer B are exchanging peer information. (b) Afterwards, peer A and peer B can connect to the newly discovered peers. Note: The two exchanged peer lists do not have to be sent in tandem.

Since PEX was not designed to be used stand-alone, no attention was paid to the bootstrapping problem. Furthermore, the cost of running PEX is low and grows only with the size of the peer's neighbourhood set (which is usually bounded). Churn is not an issue since the data in a PEX message is fresh. Only when you save received PEX messages for later use, it will become a problem as data becomes stale.

Security is not a large problem in PEX. While attackers can push fake peer lists via PEX, the information will not be propagated by good peers. The only possible security issue is that fake peer lists can be used for a DDoS attack with the multi-fake technique. Certain BitTorrent clients, e.g. Tribler [38], only use a small fixed number of addresses from a PEX message to connect to, mitigating the problem.

Connectivity issues are not necessarily taken into account. BitTorrent clients could only include neighbouring peers that are known to be connectable in PEX messages they send, but the protocol does not mandate it.

Chapter 3

PEX Behaviour Study

We have conducted a study to understand the reliability and usability of PEX messages. Our motivation is that PEX is already a critical and widely used technique for swarm discovery. We have found that there are significant inefficiencies in the currently used implementations. In this chapter we will discuss and quantify these inefficiencies.

Section 3.1 describes the crawler software we have developed and in which BitTorrent swarms we have conducted our experiments. In Section 3.2 we visualize the crawled swarms in order to find peculiarities in the topologies. Next, we focus on the diverse PEX implementations and the quality of the received PEX messages in Section 3.3. In Section 3.4 we discuss churn and the impact on the freshness of PEX messages.

3.1 Crawl experiment

PEX is not properly specified in any BitTorrent Enhancement Proposal (BEP).[1] This gives implementers a lot of freedom and as a result real world behaviour of PEX varies in how often and quickly PEX messages are sent and how many peer addresses they contain.

We study these behavioural apects of PEX by contacting peers in BitTorrent swarms and recording information about the messages received as follows. Our crawler software connects to a peer in a BitTorrent swarm and waits idly for an incoming PEX message for at most 2 minutes. When our crawler receives a PEX message or the time is up, it closes the connection. If a peer does not support PEX messages we immediately close the connection, since it would be pointless to wait for any. For each contacted peer, our crawler logs the following information:

- The start and end timestamps of the connection.
- The version of the peer's client.
- The timestamp of the first PEX message, if received.

[1] `http://www.bittorrent.org/beps/bep_0000.html`

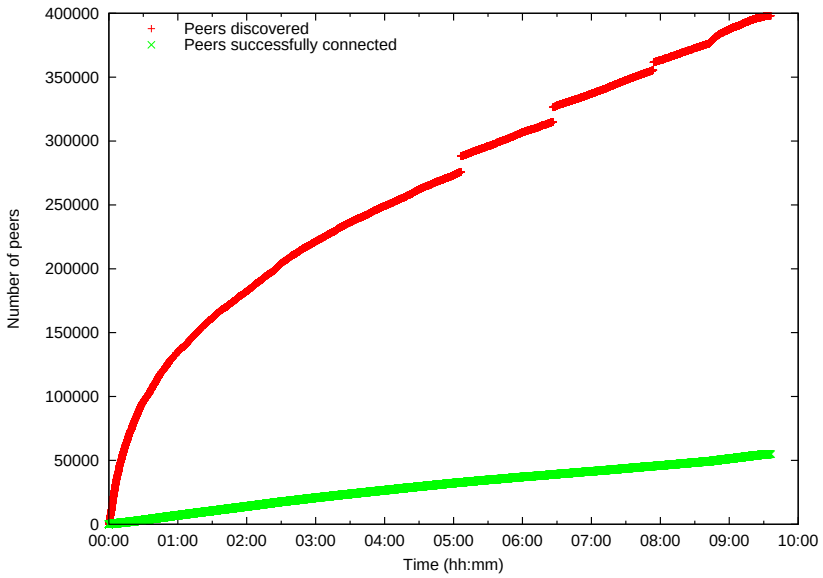


Figure 3.1: Number of peers found and connected to in the public tracker crawl.

- The peer addresses stored in this PEX message, if any.

Our crawler uses the tracker to find a set of initial peers to contact. Peers discovered through PEX are also contacted by the crawler. For each peer a connection is attempted at most once.

In addition to crawling through the swarm and recording PEX behaviour, our crawler continues to check the reachability of each peer it has contacted every 10 minutes until the end of the experiment. The first –as we dub it– "online check" is performed 5 minutes after the connection has been closed. This information is also logged and should give use an idea of the churn in a swarm and how quickly information received via PEX decays.

We have performed two experiments. We crawled three Linux distribution swarms (Ubuntu, Gentoo and Fedora) in early September 2009 and the ten largest swarms of a large public tracker in late September 2009. In the first experiment, we let the crawler run for two hours. It completed the crawl in 12 minutes, i.e. the crawler found and attempted to connect to all discovered peers in these 12 minutes. Note that peers that arrived in the swarms after these 12 minutes are not included in the crawl, since the trackers are only queried once at the start of the experiment and at most one PEX message is received from each discovered peer. In the remainder of the experiment the crawler was only checking the reachability of contacted peers. In the second experiment, the crawler ran for 20 hours and completed the crawl within 10 hours. However, we noticed that the crawler failed to crawl one of the ten swarms because none of the reachable initial peers supported PEX.

Figure 3.1 shows the number of peers crawled over time of the latter experiment. On average, the crawler connected to 1.6 peers per second during the run. It discovered peers at much higher rate, but never managed to connect to them all. Since time constraints were not an issue, we must conclude that a large fraction of discovered peers was unresponsive. We investigate this matter in the subsequent sections.

3.2 Visualization of swarms

In this section we use two visualization techniques. We visualize the Gentoo and Ubuntu swarm as undirected graphs, where a link between two peers means that they are neighbours according to a PEX message. We have omitted the Fedora swarm as its graph resembled the graph of the Gentoo swarm. The public tracker swarms were too large to be visualized as undirected graphs. Instead, we use these swarms to determine the popularity of different BitTorrent implementations in use and we visualize the connectivity through adjacency matrices.

Undirected graphs

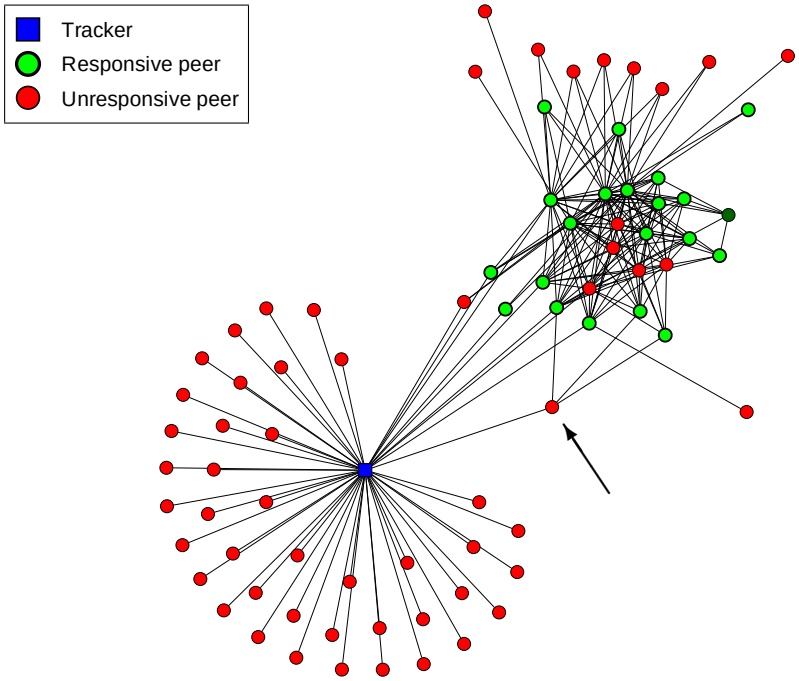


Figure 3.2: Crawl of the Gentoo swarm.

The graph of the Gentoo swarm (Figure 3.2) shows that a significant number of peers received from the tracker are not responding to a BitTorrent handshake. These peers may have gone offline, yet have recently announced their presence to the tracker, or they are behind a NAT. For a few peers, the latter is clearly the case, e.g. the peer marked by an arrow in the figure. This particular unresponsive peer was not only known by the tracker, but was also mentioned by three other peers via PEX. Assuming peers do not lie about who their neighbours are, this indicates the unresponsive peer is firewalled or behind a NAT. In general, the quality of a tracker response seemed to be mediocre. In the public tracker swarms, between 30% and 56% of the peers were reachable, with a median of 39%. The Fedora swarm had

the worst tracker response: only 1 out of 44 peers received from the tracker were reachable.

The quality of the tracker response for the Ubuntu swarm was exceptional, as 48 of the initial 50 peers were responsive. The graph of this swarm is shown in Figure 3.3. We can see that a large central cluster exists, consisting of a mix of responsive and unresponsive peers. Remarkable are two smaller seperated groups of peers left and below this cluster with a high number of unresponsive peers.

The crawl logs tell us that the left cluster (Figure 3.4a) consists of Transmission BitTorrent clients. They claim to be connected to unresponsive peers that all have an IP in the 0.x.x.x, 1.x.x.x, or 2.x.x.x range.[2] These address ranges were either private or unallocated at the time of the crawls [3] and are useless for other peers. This means the PEX implementation of certain Transmission versions is faulty. The crawl logs indicate that at least versions 1.74 and 1.75b2 seem to be affected.

The bottom cluster (Figure 3.4b) 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 connectable. Looking at the source code of Deluge [16], we see that it binds to the libtorrent library [23]. This BitTorrent implementation only includes peers in PEX messages that are found to be connectable and hence are very unlikely to be firewalled. 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. Oddly enough, these peers are only reported in PEX messages sent by these Deluge peers. We can only conclude that these peers may run a modified version of Deluge.

Whether the faulty PEX implementation of the Transmission clients and the peculiar behaviour of these Deluge clients have a large impact depends on the popularity of these clients. According to the top 10 popular clients shown in Figure 3.5 the impact should be small. Transmission has a small market share and Deluge is not even in the top 10 (it is ranked #19).

Perhaps more worryingly is the existence of clients that report to support PEX, yet never seem to send any PEX messages. These clients leech in a sense on other PEX supporting clients. They can freely use received PEX messages without ever having to reciprocate. Azureus' behaviour, however, is caused by a bug. The code responsible for creating PEX messages is unreachable due to uninitialized fields [4].

Adjacency matrices

The connectivity in a graph can be visualized by taking the graph's adjacency matrix and plotting the non-zero elements of that matrix. We will discuss a single swarm of the public tracker crawl as the other swarms exhibit similar visual features. The adjacency matrix of this swarm is shown in Figure 3.6. The peers are

[2] Peers with a 0.x.x.x IP address were not drawn to reduce visual clutter.

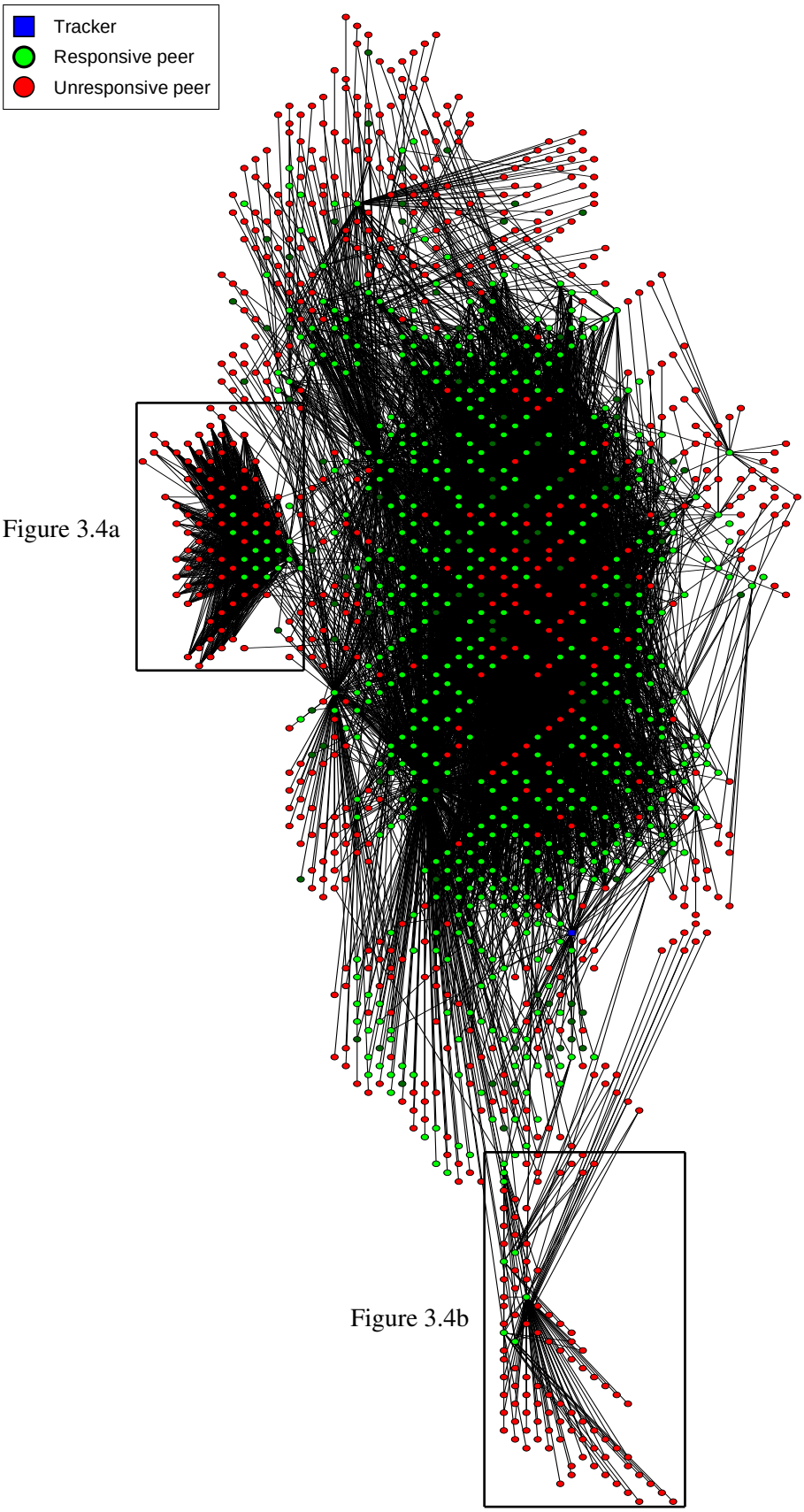
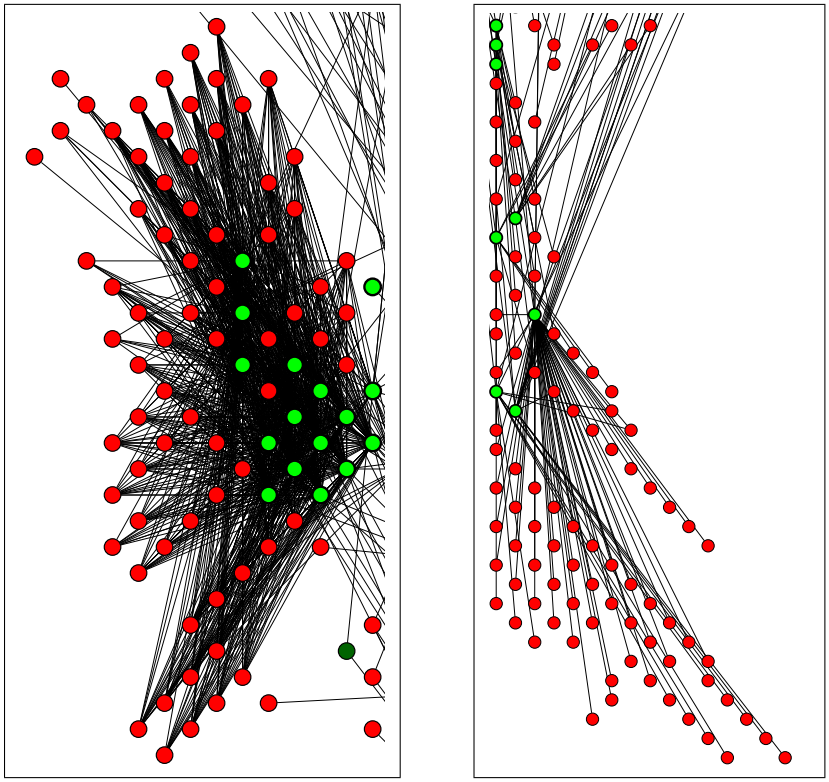


Figure 3.3: Crawl of the Ubuntu swarm. Details in Figure 3.4.



(a) Transmission cluster (b) Deluge cluster

Figure 3.4: Unresponsive peers in the Transmission cluster have addresses in private or unallocated IP ranges. The unresponsive peers in the Deluge cluster do not belong to a particular IP range.

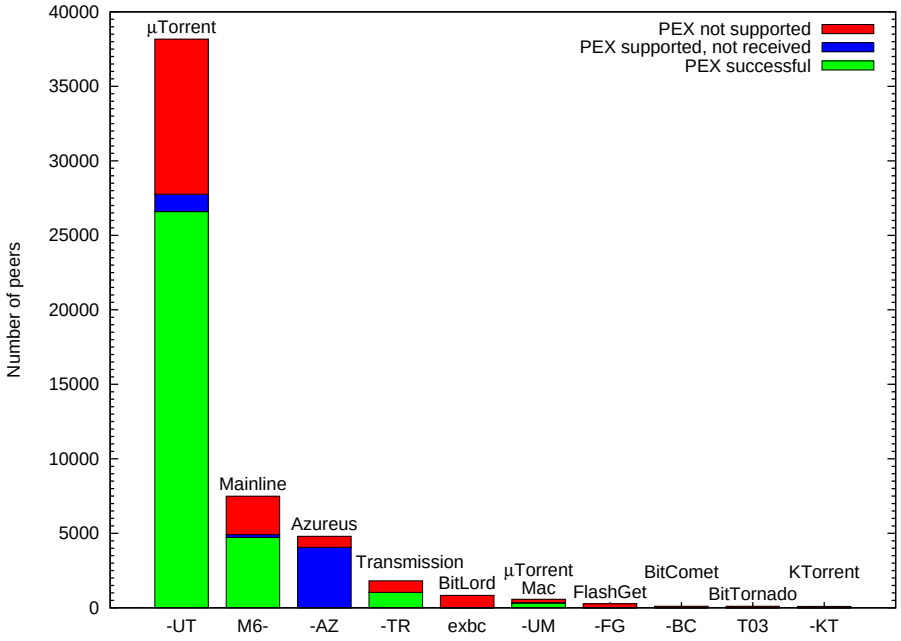


Figure 3.5: Top 10 BitTorrent clients encountered during the public tracker crawl.

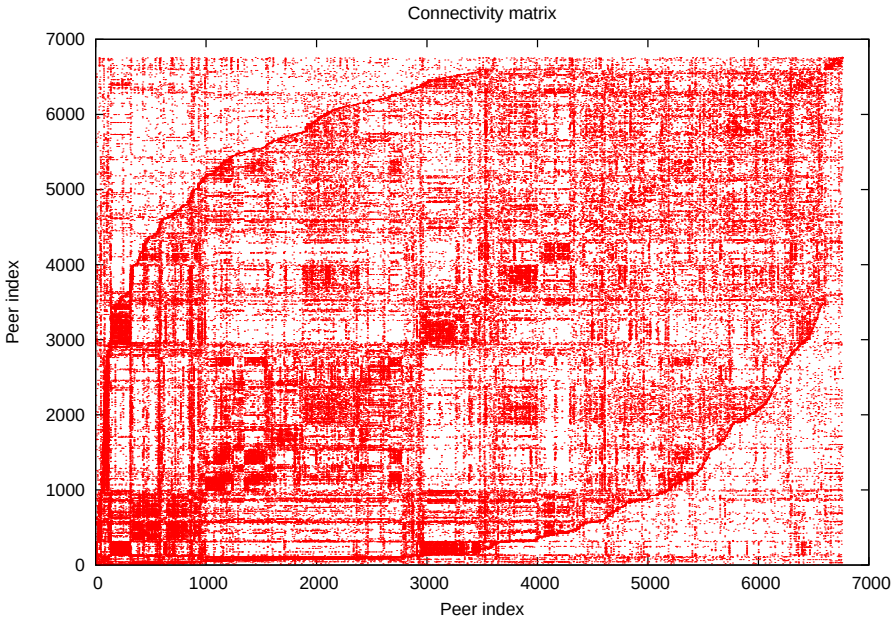


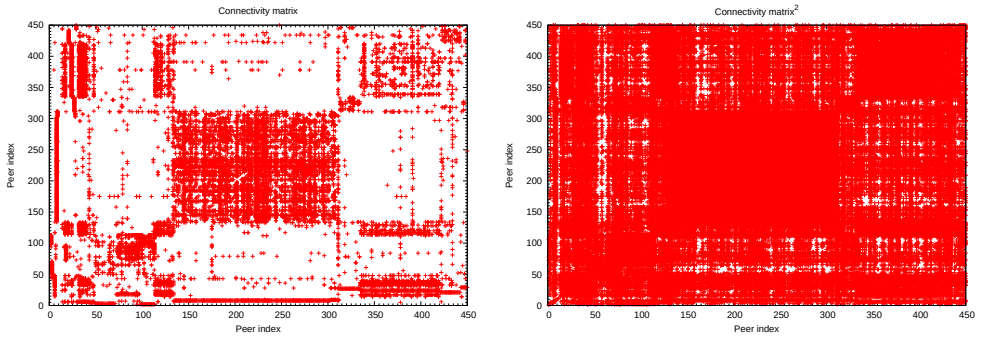
Figure 3.6: The connectivity matrix of responsive peers in public swarm #10. A dot at (i, j) means that i and j are claimed to be neighbours.

ranked by connection timestamp, i.e. a peer A has a lower index than peer B when our crawler connected to peer A before peer B.

We have tried to compare this matrix to connectivity matrices found in other works [2, 15], but we were unable to do a valid comparison. The peers in other works are ranked by the time at which they join a swarm. Since in our experiment we do not know when a peer has joined a swarm, we cannot rearrange the peers in our adjacency matrix in a similar way. Even if we could rank the peers similarly, a valid comparison would still be difficult:

- The matrices found in other works are a result from simulations with mostly homogeneous peers, while our matrix is the result of crawling real world swarms of heterogeneous peers.
- The matrices found in other works are static snapshots taken at a single point of time. Our crawl inevitably spans a longer time frame.
- Matrices found in other works are true adjacency matrices, while our matrices are constructed using PEX messages under the assumption that peers only list other peers that they are actually connected with, but that they do not list peers they have only heard of. This assumption is valid for major open source clients, but we will see it does not hold in general.

Still, our matrices display visual patterns that resemble the ones that can be found in [15], albeit less prominently. The authors of [15] run a simulation with



(a) Path lengths $= 1$ (b) Path lengths ≤ 2

Figure 3.7: The connectivity of the first 450 encountered peers in the same public swarm as in Figure 3.6.

a modified tracker that assigns each peer to a certain clique. This modified tracker creates swarm topologies that consist of many clusters with good inter-cluster connectivity. These clusters appear in adjacency matrices as tight boxes. Inter-cluster connectivity appears as dots outside these boxes, indicating two peers from different clusters are connected.

We can see that our connectivity matrix in Figure 3.6 contains box patterns, indicating the existence of clusters. The existence of these clusters, but also of inter-cluster connections, is important for faster swarm discovery as the path length between any two peers is greatly reduced when there are peers in a cluster with connections to other clusters. Such peers serve as a gateway to other clusters. The path length between any two peers in two connected clusters would be at most 3 hops: the path consists of at most two cluster-gateway peers, and the final destination.

We can find the number of paths of length k or less between peers i and j by taking the adjacency matrix to the kth power. Figure 3.7 shows the adjacency matrix A for the first 450 encountered peers in the crawled swarm and the corresponding A^2 matrix. The dense scatter plot of A^2 suggests most of these 450 peers are within a 2 hops distance. However, if we use $d(i, j)$ to denote the distance between peers i and j, then for this swarm it actually holds that

$$P(d(i,j) \leq 2 \mid i,j \leq 450) = \frac{91,891}{450^2} = 45\%,$$

where 91,891 is the number of nonzero elements in A^2. We computed this metric for nine of the ten public swarms[3] for values of k up to 5 (Table 3.1). Swarms with more paths of length 3 or shorter, i.e. a higher probability $P(d(i, j) \leq 3)$, show the tight-boxes pattern more clearly in their connectivity matrices, with the notable exception of swarm #3. Instead of clusters, there seem to be two peers

[3]The crawler failed to thoroughly crawl Swarm #5 as discussed in Section 3.1.

Swarm#	n peers	$P(d(i,j)=1)$	$P(d(i,j)\leq 2)$	$P(d(i,j)\leq 3)$	$P(d(i,j)\leq 4)$	$P(d(i,j)\leq 5)$
1	7428	0.52%	22%	88%	99.9%	99.99998%
2	5589	0.10%	1.8%	13%	53%	89%
3	4619	0.23%	56%	95%	99.9%	99.9998%
4	9169	0.16%	5.0%	40%	87%	99%
*5**	*15*	*0%*	*0%*	*0%*	*0%*	*0%*
6	5973	0.19%	6.7%	40%	92%	99.8%
7	5236	0.36%	65%	97%	99.9%	99.96%
8	4393	0.21%	4.3%	31%	83%	99%
9	5525	0.33%	8.8%	59%	97%	99.96%
10	6761	0.43%	17%	82%	99%	99.9%
Total	54708					

Table 3.1: The distance $d(i,j)$ between any two peers i and j in the public tracker swarms.
* *The crawl failure of Swarm #5 is discussed in Section 3.1.*

which claim they are connected to nearly all other peers. These two peers run an Opera BitTorrent client and will be discussed in the next section. While not every swarm shows signs of strong clustering, we do see that in at least 89% of the cases the path between two random peers consists of at most 5 hops in the swarm graph constructed by crawling, and hence, the majority of the swarm can be discovered using few PEX messages.

3.3 Poor quality of PEX messages

During the experiments we found significant differences in the behaviour and the quality of information in PEX messages. This observation has never been published before. The variation can be attributed to PEX not being properly specified in any BitTorrent Enhancement Proposal and thus 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 data from our largest crawl, i.e. the public tracker swarms. In the subsequent probability density figures, we only include the data of the 4 most widely used, PEX supporting clients to reduce visual clutter: µTorrent, Mainline, Transmission, and µTorrent Mac.[4]

Arrival and size

The first significant difference in behaviour is how quickly a BitTorrent client sends a PEX message after the connection has been established (Figure 3.8). Transmission tends to send its first PEX message almost immediately after the handshake: 75% of the messages arrives within 4.5 seconds. The time it takes for the other clients to send the first PEX message appears to be more or less uniformly distributed. As a result, 75% of the messages from e.g. µTorrent arrives after 19.5 seconds, with an average arrival time of 35.0 seconds. A possible explanation for this "uniform distribution" could be that these clients use a global timer to send periodical PEX messages, and therefore the time it takes to receive the first PEX message depends on the time at which you connect to them. Since most messages arrive within a minute (for e.g. Mainline, 95% of the messages), it is likely that this global timer ticks every minute. The small fraction of the messages that arrives after more than a minute are possibly delayed by congested uplinks.

There also seem to be two different kinds of behaviour when it comes to the number of peers listed in a PEX message. Figure 3.9 shows the distribution for Transmission and µTorrent. The Mac version of µTorrent and Mainline behave quite similarly to the latter. Transmission appears to enforce a strict upper bound of 50 peers in a PEX message, with the exception of version 1.11 and older. The older versions of Transmission constitute less than 0.5% and the largest message encountered contained 81 peers.

[4] Azureus is not included as it does not send PEX messages as discussed in Section 3.2.

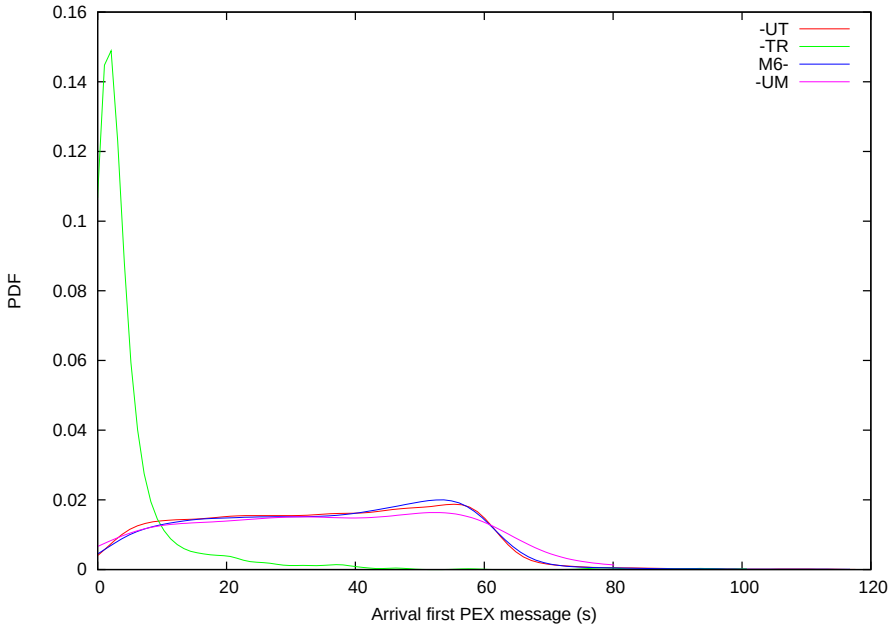


Figure 3.8: Empirical density function of the arrival of the first PEX message.

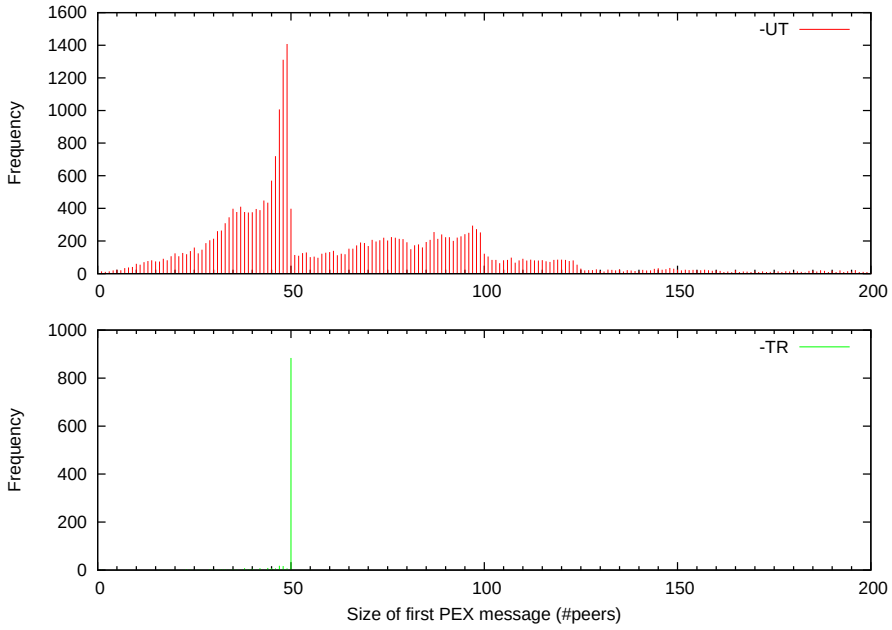


Figure 3.9: Size of the first PEX message histogram.

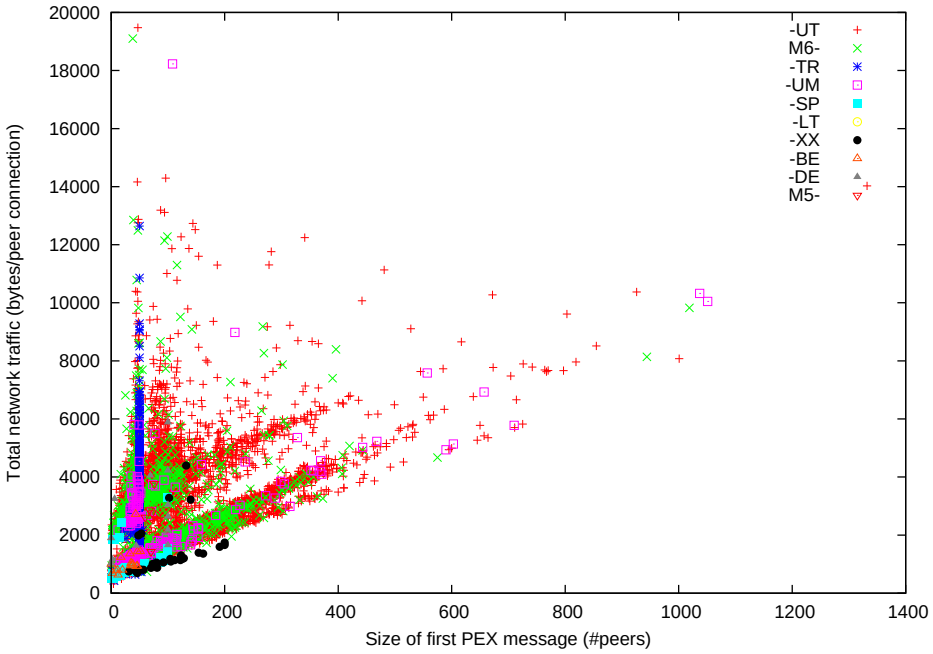


Figure 3.10: Number of peer stored in a PEX message versus the number of bytes received and sent to communicate with the PEX message's sender.

The distribution of the PEX size for µTorrent varied much more. Obviously it uses a higher size limit, if any. With a higher limit, the variation is natural as the size of a peer's neighbourhood set becomes a more influencing factor. Unfortunately, it is hard to determine what size limit µTorrent uses. While 95% of the messages contained at most 177 peers, we did find (albeit a few) larger messages. Less than half a percent of the messages was larger than 362 peers, with the largest message containing 1332 peers. This largest message was sent by a more recent version of µTorrent (version 1.9) compared to the older most common version (1.8.3). Small PEX messages fortunately rarely occur. Less than half a percent contain less than 5 peers. The same holds for Transmission.

Communication cost

We also found that the total amount of network traffic that was generated while waiting for a PEX message to arrive varied. This communication cost is of interest when you start a BitTorrent connection for swarm discovery purposes and not for bartering. Since this communication cost certainly depends on the size of a PEX message, we plotted the communication cost against the size of the received PEX message in Figure 3.10. This should give us an overview of traffic generated by non-PEX messages.

In this figure we have omitted PEX messages received from Opera clients. We

found that these messages were extremely large compared to other messages received from other implementations. They contained tens of thousands of peers, with the largest received message listing 91,145 peers costing us 638,708 bytes of downstream traffic. It is not likely that Opera sends its full neighbourhood set in its first PEX message, since not many hosts would be capable of maintaining 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 are very likely to be offline. Another explanation is that it may send a list of peers it has heard of and this would be a serious security flaw. The connectivity matrices of the swarms in which we encountered Opera clients tell us that the clients claim to be connected to nearly all other responsive peers.

Omitting Opera, Figure 3.10 shows that it costs us at most 4 KB of data to receive a PEX message in 95% of the cases. 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. Transmission shows this variance clearly for its messages of size 50. When we compare Transmission to µTorrent, we find that the latter is generally more bandwidth efficient. 95% of the PEX messages sent by µTorrent costs us at most 4 KB, while Transmission needs at most 5 KB.

Responsive fraction

In Section 3.2 we already discovered that not all discovered peers were not responsive. Since the tracker was only used initially, most unresponsive peers were reported in PEX messages. Figure 3.11 shows rather unsatisfying results. On average, between 19.5% and 21.7% of the peers listed in a PEX message is responsive depending on the client version. At most 48% of the peers listed in 95% of the messages sent by µTorrent and Transmission is reachable. The peak in the probability distribution gets narrower and moves to the left if we consider how many peers also report to support PEX. It gets even worse if we also consider whether they actualy send a PEX message which is important for swarm discovery. On average, only 14% of the peers in messages sent by µTorrent are useful for discovering more peers, i.e. they are connectable and would send us a PEX message. For Transmission this percentage is 11%. More shockingly, 18.6% of the PEX messages sent by Transmission are completely useless: none of the peers are reachable or send a PEX message within 2 minutes. For µTorrent, this percentage is only 7.2%. The large difference can be attributed to the existence of Transmission clients with a buggy PEX implementation discussed in Section 3.2.

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 3.12, however, shows that only 5% of the unresponsive peers are mentioned more than 22 times. This might

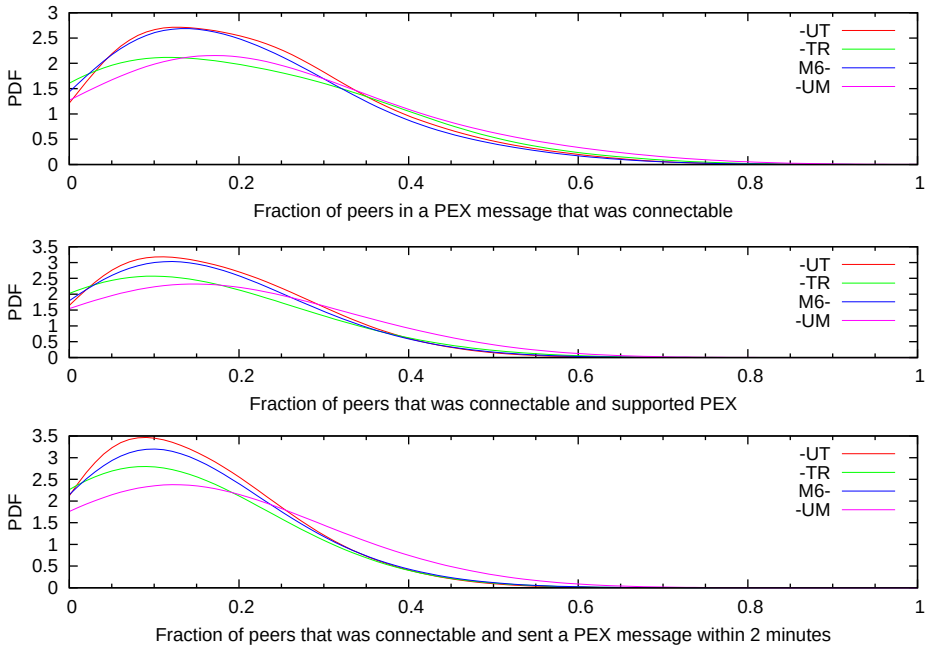


Figure 3.11: Quality of PEX messages.

indicate that most of the peers are firewalled, but keep in mind that our crawler only collected at most 1 PEX message from each peer, which could have skewed the distribution. Additionally, peers that are only mentioned a few times may still be congested if they were active in other swarms we have not crawled. Therefore, we cannot deduce anything about a peer's reachability based on the number of times it is mentioned.

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. These peers are often called *remote peers* and *local peers*, respectively. Peers you are connected to via an outgoing connection are more likely to be also connectable for others than peers you are connected with via an incoming connection. As mentioned before, the libtorrent library [23] chooses to only include local peers in its PEX messages.

3.4 Remaining uptime of peers

For improved swarm discovery, we need to determine how quickly information received from PEX messages decays. Since our crawler attempted to contact each peer it saw in a PEX message and regularly checked their reachability, we can compute the decay rate. Note that the crawler also continued to check the reachability of a peer after an unsuccessful online check. We will, however, only consider the

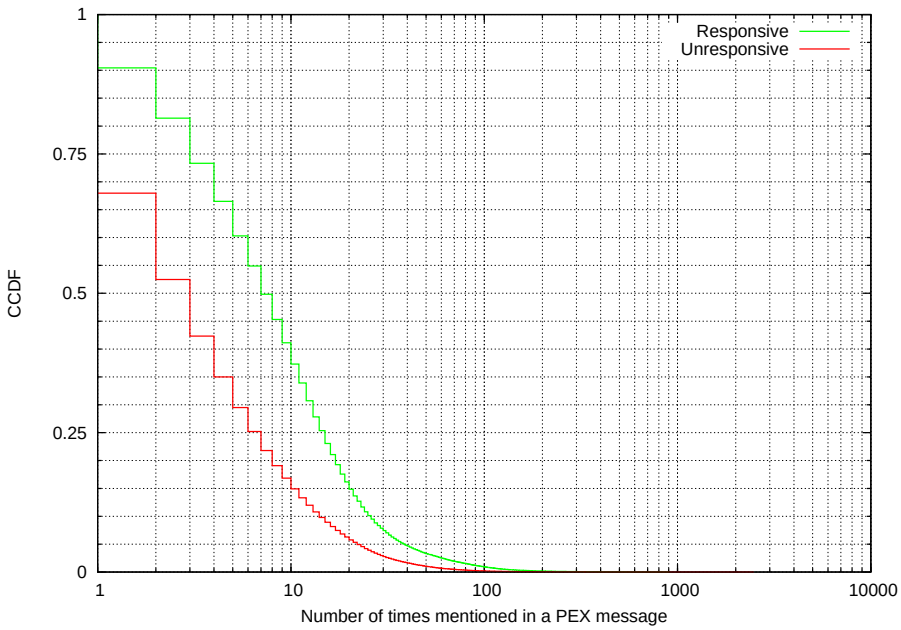


Figure 3.12: Distribution of the number of times responsive and unresponsive peers are mentioned in a PEX message.

successful checks up to that first unsuccessful check. This means we underestimate the remaining uptime of a peer. For example, it is possible that a peer was online during 10 checks, but failed the 6th check for some reason. In this case, we would estimate its uptime to be 5 periods instead of 10. We justify this underestimation: When you want to join a swarm, you want a swarm peer to be reachable right now, not several minutes into the future.

Hence, we define the remaining uptime of a peer to be the difference between the timestamp of the last successful online check not preceded by an 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. Additionally, we only consider the uptimes of peers we have connected to in the first half of the measurement. This provides equal opportunity to observe remaining uptimes less than $\frac{\tau}{2}$, where τ is the length of the measurement [34].

Figure 3.13 shows the overall remaining uptime. We can see that a quarter of the peers no longer responds after 25 minutes and that after 105 minutes only half remains. This means that if we want to know at least n responsive peers 25 minutes in the future, we should at least now know $(1 - \frac{1}{4})^{-1}n = \frac{4}{3}n$ responsive peers.

The shape of the plot suggests an exponential distribution at first sight. Unfortunately, we were unable to find a suitable value for the rate parameter λ. Only for uptimes larger than 20.000 seconds (5 hours and 33 minutes), the plot seems to follow an exponential distribtution with $\lambda = \frac{\ln 2}{8000}$, i.e. a median of 8000 seconds or 133 minutes. However, empirical data shows that the median is actually 105

minutes, and hence, using $\mathrm{Exp}(\frac{\ln 2}{8000})$ as a model would be too optimistic. Instead of finding a suitable distribution, we have outlined the uptime percentages up to two hours in Table 3.2. This table will be used later in Section 5.4.

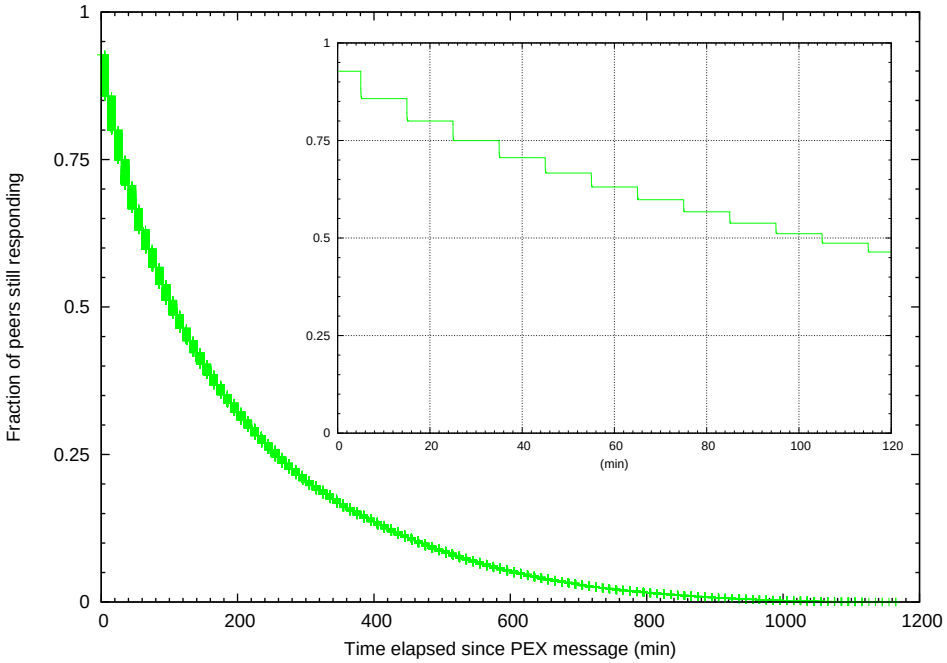


Figure 3.13: Overall remaining uptime of peers in the public tracker crawl. Inset shows a close-up of the first 2 hours.

Uptime larger than t (min)	Fraction of peers
0	92.7%
10	85.7%
20	80.0%
30	75.0%
40	70.6%
50	66.7%
60	63.1%
70	59.8%
80	56.7%
90	53.8%
100	51.1%
110	48.7%
120	46.4%

Table 3.2: Uptime lookup table for the first 2 hours of Figure 3.13.

Chapter 4

2-Hop TorrentSmell Design

In this chapter we present our swarm discovery algorithm called 2-Hop TorrentSmell based on our findings from the previous chapter. The algorithm consists of two parts and an overview of the algorithm is given in Section 4.1. The first part is the extended keyword search for finding the distributed trackers via Tribler's overlay network and is described in more detail in Section 4.2. The second and core part of the algorithm is the actual tracking of swarms via our pull-based RePEX tracking algorithm and is described in Section 4.3. We discuss which of the design requirements from Chapter 2 our algorithm meets in Section 4.4.

4.1 Overview

For swarm discovery it is important that you are able to find someone who has information about the swarm you are interested in, in order to connect to the peers in that swarm. In the case of central trackers, that someone is statically known. In the case of the DHT, there is a lookup mechanism to find a node responsible for maintaining the peer list.

For both the central tracker and the DHT nodes, it is important that they can track a swarm. This is done by push-based system, where members of the swarm announce their presence to the tracker or the corresponding DHT node.

Similarly, our 2-Hop TorrentSmell algorithm can be divided into two important parts. Finding a peer who has information about a swarm is done using an extended version of Tribler's fast zero-server keyword search (Section 4.2) and tracking a swarm is done using our RePEX algorithm (Section 4.3). The key idea in 2-Hop TorrentSmell is that peers that have recently downloaded a file most likely have more information about that file. Contacting those peers would be a good idea to get closer to the swarm and resembles how ants follow a *smell gradient*.

Luckily, in Tribler this information is already available. Tribler Peers periodically share their preferences, i.e. a list of their latest 50 downloads, using the BuddyCast protocol [31]. Information received via BuddyCast is stored in a peer's MegaCache [30]. With just a few modifications, the existing keyword search [40]

can be used to find peers who have recently downloaded a file by including them in the search results. In our 2-Hop TorrentSmell algorithm, Tribler peers who have downloaded a file will remember who were in the swarm. Since this information will become old, they periodically refresh this information using the RePEX tracking algorithm.

When the extended keyword search and the RePEX algorithm are in place, we can join a swarm using 2 hops:

1. Perform a keyword search and select a file to download from the search results.
2. Hop 1: Contact the peers listed for the swarm of the selected search result. They should have recently downloaded that file and should be tracking the swarm. Contacting them should give us a list of swarm peers.
3. Hop 2: Connect to the peers in the received list of swarm peers.

A pictorial overview of the 2-Hop TorrentSmell algorithm is given in Figure 4.1.

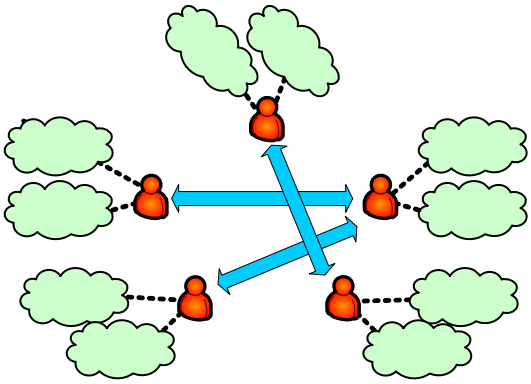
4.2 Extending Tribler's keyword search

The 2-Hop TorrentSmell mechanism of finding tracking peers is integrated with the existing keyword search. Our rationale is that when a user is searching for content and selects a file to download, that download should start immediately. It helps when a list of tracking peers is included with each search result, such that when a user chooses a file to download, it does not first have to find a tracker. This reduction of response time is especially needed in applications like video streaming for improved user experience.

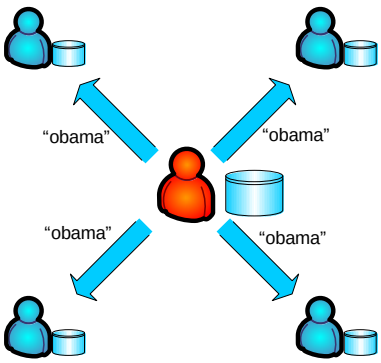
The current version of keyword search in Tribler uses two messages. The query message is simple and only contains a string of keywords. The query reply message contains a dictionary mapping infohashes to various metadata, including the torrent name and classified category [20]. Upon receiving a search query, Tribler peers currently look for content in their MegaCache that match the keywords and simply send the results back.

In the extended keyword search for 2-Hop TorrentSmell, the metadata stored in the dictionary of a query reply message is extended with two fields: "peer type" and "peers". Upon receiving a search query, we still look for matching content in the MegaCache, but now will do the following for each search result:

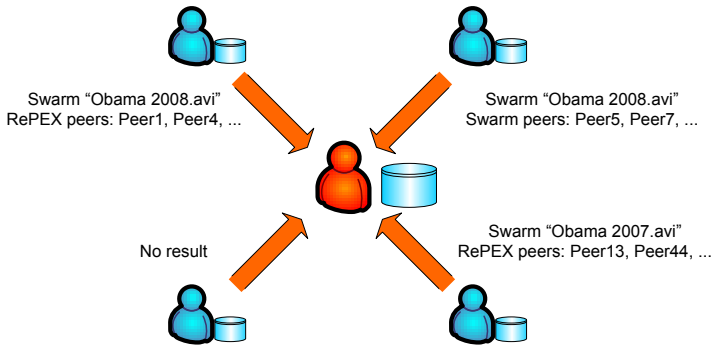
- If we are not tracking this swarm, find all peers that have downloaded this file according to our MegaCache. Use this list of peers for the "peers" field and set the "peer type" field to "RePEX".
- If we are tracking or are still seeding this swarm, set the "peer type" field to "Swarm" and the "peers" field to our peer list (set K_1 from the algorithm in Section 4.3).



(a) Tribler peers track the swarms of their last 50 downloads using RePEX. A peer's last 50 downloads are gossiped through BuddyCast.



(b) A Tribler peer issues a keyword search for "obama" to its neighbouring peers in the Tribler overlay.



(c) The neighbours reply with peers found in their databases of stored BuddyCast messages and which are known to be tracking swarms that match the keywords. In case a neighbour is tracking a matched swarm, it will reply with swarm peers instead.

Figure 4.1: Overview of 2-Hop TorrentSmell.

By having tracking peers reply with swarm peers instead of itself as a tracker peer, we save an extra roundtrip. This also saves us from having to introduce a new message for querying tracking peers for a peer list.

4.3 Tracking the swarm using RePEX

As mentioned in Section 4.1, the tracker and peers stay in touch using a push-based design. Hence, it is the peer who is responsible for regularly informing the tracker of its presence. As discussed in Chapter 2, the tracker does not know whether the peer is connectible and whether it is actually a member of the swarm without performing a check. The same situation applies to the DHT solution.

We instead propose an algorithm, called RePEX, that puts the responsibility on the tracker to contact the peers. This idea is not new. For example, in [19] so called *entry points* are continously exploring the swarm. These entry points, however, are also member of the swarm themselves. Peers running our RePEX algorithm on the other hand are not continuously available in the swarm they are tracking.

In our RePEX algorithm, whose name is a contraction of the words 'reconnect' and 'PEX', a tracking peer maintains a cache of previously encountered swarm peers that support PEX, for each of its last 50 downloads (the same number of downloads as is gossiped via BuddyCast), and periodically reconnects to these peers to check whether they are still reachable. During the check, the tracking peer waits for a PEX message to build a list of secondary peers. These secondary peers will be checked in the event one of the primary peers no longer seems reachable.

Algorithm details

For a given swarm, the RePEX algorithm creates and maintains a *SwarmCache*: a set K_1 of primary peers we know directly, and the sets K_{2p} of secondary peers we know indirectly through these peers, but are directly known by peer $p \in K_1$ (Figure 4.2). These sets are restricted in *size* by configurable parameters S_1 and S_2:

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

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*. A PEX ping is the act of 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.

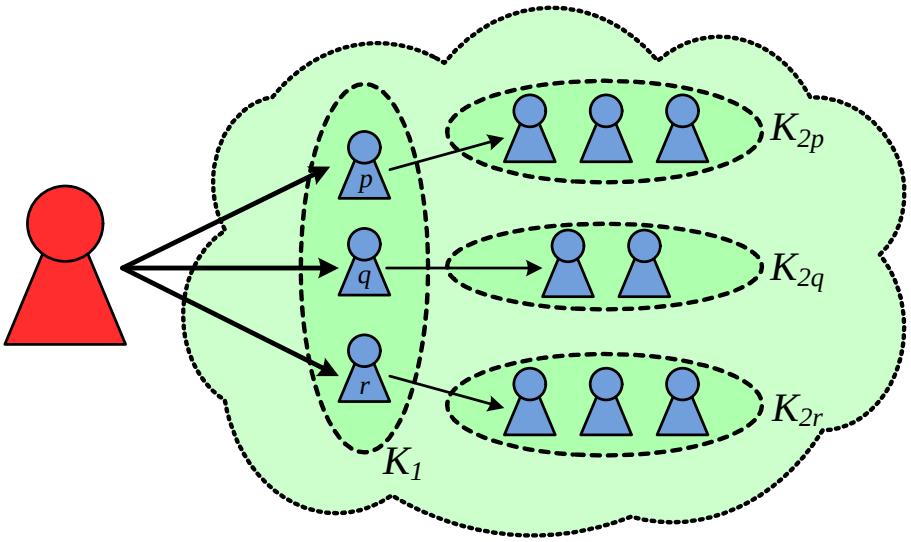


Figure 4.2: A *SwarmCache* consists of directly known primary peers (K_1) and indirectly known secondary peers (K_{2p} for each $p \in K_1$).

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* whether a peer is alive, 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 we have found (at least) S_1 peers that are alive or until we have checked all known peers, i.e. until the following condition holds:

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

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 (e.g. 2-Hop TorrentSmell itself) or a central tracker, to get a set of peers B. PEX ping

the newly discovered peers minus the ones we have already pinged ($B \setminus K$) until we have found sufficient peers or have checked all known peers, i.e. until the following condition holds:

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

4. Construct a new set K'_1 of connectable PEX supporting peers using all peers known to be alive, preferring peers from K_1. That is, for some chosen set $\hat{C}$, $K'_1 = A(K_1) \cup \hat{C}$, where

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

5. Each new set K'_{2p} of secondary peers is constructed by choosing a subset of peers received during the PEX ping. Formally, for each $p \in K'_1$ and for some $\hat{R}_p$, $K'_{2p} = \hat{R}_p$, where

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

4.4 Requirements discussion

Our proposed swarm discovery solution meets most of the requirements from Section 4.4. It fails to meet the security requirement, but we will discuss how our algorithm can be made more secure.

Since 2-Hop TorrentSmell algorithm is designed for Tribler, we will ignore the bootstrapping required to join the Tribler overlay network,[1] but note that it is required to be in the Tribler overlay for the keyword search to work. For executing the RePEX algorithm for the first time, no bootstrapping is needed. Since a Tribler peer executes this algorithm after it has downloaded a file and left the swarm, it can use the peers in its neighbourhood set from when it was still in the swarm.

Our algorithm is scalable. Since each Tribler peer will start tracking a swarm using the RePEX algorithm after it is done downloading, popular swarms will automatically give rise to more distributed trackers over which the load can be distributed. Lookups in our algorithm are also fast. Existing keyword search queries already have a median response time of 325 milliseconds [40]. We expect that retrieving a peer list for a swarm should take less than a second, depending on the number of search queries required, but further work is needed to implement and test the extended keyword search component.

Still, there is room for improvement. Consider the scenario of a new, but popular swarm. Since it is a new swarm, not many people have completed downloading the

[1] Bootstrapping in Tribler is discussed in [30].

file, and hence, there are not many distributed trackers yet. As a result, the few trackers can receive many peer list requests. The Tribler peers that are currently in that new swarm, however, could perfectly well act as distributed trackers as well. Unfortunately, it is not known by others that these downloading Tribler peers are in that swarm. Since they are not known, they will not receive any peer list queries. A possible solution could be to introduce a new gossip message, such that the downloading Tribler peers can announce that they are downloading a particular file.

Our algorithm does not take security directly into account. However, if there is a reputation function that can rank peers on their perceived trustworthiness, then we can make 2-Hop TorrentSmell more secure by preferring more trustworthy peers to query. An example of a reputation system is BarterCast, which is based on bandwidth contribution. BarterCast is used to effectively prevent freeriding behaviour [26].

Unlike current solutions, our algorithm is effective and takes connectivity issues into account. The pull-based RePEX algorithm only keeps reachable, and hence, connectable peers in the SwarmCache. Whether churn is effectively dealt with, depends on how often the SwarmCache is refreshed, but also on the chosen parameters S_1 and S_2. We will investigate this in the next chapter.

A pull-based design, in addition to being useful for checking reachability, also has a deployability advantage. The central tracker and DHT solutions work because there are sufficient BitTorrent clients supporting it and are announcing their presence. Since RePEX peers are using the widely deployed PEX protocol to track a swarm, our solution does not require that other BitTorrent clients support 2-Hop TorrentSmell.

Chapter 5

Implementation and Evaluation

We have implemented the RePEX algorithm discussed in the previous chapter and added it to the main development branch of Tribler. An instrumented implementation was deployed with the beta release of Tribler 5.2 in late February 2010. In Section 5.1 we describe the current implementation and how it fits in the current architecture of Tribler. We discuss the performance of our RePEX algorithm in Section 5.2 using the top 10 swarms of a popular BitTorrent tracker and the Ubuntu swarm. To see how well the algorithm would perform in other swarms, we have collected measurement data from users who were running Tribler 5.2b. The results of the deployed algorithm is discussed in Section 5.3. In Section 5.4, we evaluate the algorithm and consider possible improvements.

5.1 RePEX implementation

The part of the Tribler Core responsible for BitTorrent downloads/uploads can conceptually be split into two components: Download and BT Engine. A Download is the representation of a single running BitTorrent download/upload and is responsible for starting and stopping the BT Engine. The BT Engine is responsible for connecting to peers and tasks like requesting file pieces.

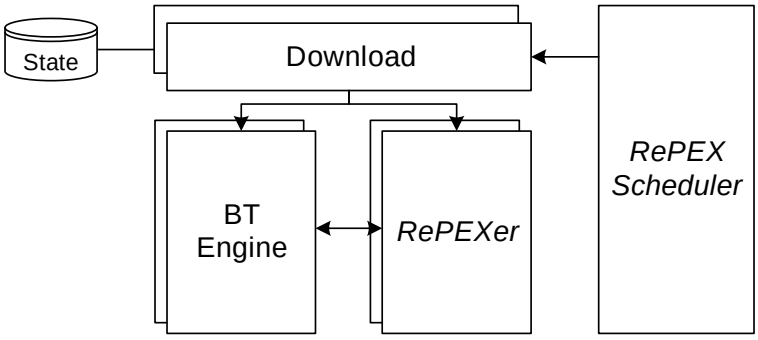


Figure 5.1: Simplified overview of the RePEX implementation.

For the implementation of the RePEX algorithm, we first added the RePEXer component and wired it to the Download and BT Engine components (Figure 5.1). When a Download is asked to run the RePEX algorithm, it pauses the BT Engine and creates and sets up an instance of a RePEXer. The RePEXer then supplies the BT Engine with instructions –as opposed to the BT Engine running autonomously– and the BT Engine in return reports back. The SwarmCache is stored in the corresponding Download State.

To run the RePEX algorithm periodically, we have implemented the RePEX Scheduler component that scans all Downloads for outdated SwarmCaches. When it finds an old SwarmCache, it instructs the Download to start the RePEX algorithm. After the algorithm has been executed, the scheduler continues scanning for other outdated SwarmCaches.

Key	**Value**
(IP,port)	**Key** / **Value** 'last_seen' — Timestamp when this peer was last seen alive. 'pex' — [((IP,port), PEX flags)]; Represents K_{2p}. 'prev' — Optional flag value, indicating this peer was also in the previous version of the SwarmCache.

Figure 5.2: The SwarmCache data structure.

In order to determine the freshness of a SwarmCache, we record a last-seen timestamp for each peer in the SwarmCache data structure (Figure 5.2). The SwarmCache is a dictionary representing the set K_1 from the algorithm. It associates the address of a peer p in K_1 with another dictionary containing the last-seen timestamp and a list of peers representing the set K_{2p} from the algorithm. This 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 [37], but are not used anywhere in the current implementation of our algorithm. Future versions can use these flags to add bias to the selection of peers. Optionally, the SwarmCache may contain a flag indicating whether a peer in K_1 was also in the previous version of the SwarmCache. This flag is currently only used for measurement purposes.

The cost of storing the SwarmCache data structure is as follows. Storing a primary peer costs 6 bytes for the IP address and port, 8 bytes for the timestamp, optionally 1 byte for the 'prev' flag, plus the bytes needed to store secondary peers. Storing a secondary peer costs 6 bytes for the IP address and port and 1 bytes for the PEX flags. Given parameters S_1 and S_2, the upper bound of storage costs would be $S_1(15 + 7S_2)$ bytes per SwarmCache.

To refresh a SwarmCache quickly, the implementation checks several peers in parallel. Currently, the implementation is configured to use a limit of 4 TCP sockets initially. When one of the peers failed to respond, it is allowed to use 8 TCP sockets to check secondary peers. As discussed in Chapter 3, a large fraction of these

peers (received via PEX) are likely unreachable. Using more sockets allows us to mitigate multiple timeouts accumulating in the algorithm's runtime.

The current implementation also performs filtering on the PEX messages when refreshing the SwarmCache. As we have seen in Chapter 3, certain Transmission clients send PEX messages with private or unallocated IP addresses. These IP addresses are removed from the PEX messages. The current implementation also only considers PEX messages that contain at least one IP address after filtering. If a received PEX message is empty after filtering, the peer that sent the message is not deemed to be "alive", since that peer would be useless for discovering other swarm members.

The initial SwarmCache for a swarm is constructed when we are still bartering. The existing Download component already keeps track of the current neighbourhood set. We have slightly modified this data structure and added for each peer in the neighbourhood set a PEX message counter and a flag indicating whether it is a local or a remote peer. The PEX message counter is used to tell whether a peer in the neighbourhood set has sent a PEX message. Note that only local peers are known to be connectable. Upon leaving the swarm, i.e. when the Download stops, we select local peers that have sent a PEX message from the neighbourhood set and store it in the SwarmCache.

5.2 RePEX performance

We conducted a small experiment to analyse the performance of the implementation. In January 2010, over the course of 57 hours and 23 minutes, we ran the RePEX algorithm on a single client and let it create and maintain a SwarmCache for the 10 largest swarms on a public tracker and the Ubuntu swarm. We configured the implementation to store $S_1 = 4$ peers in the SwarmCache, and $S_2 = 10$ peers received via PEX for each peer in the SwarmCache. SwarmCaches were refreshed when they became older than 10 minutes.

Refresh costs

Figure 5.3 shows the performance of the RePEX algorithm for one of the swarms. For this particular swarm we can see in the first plot that each refresh of the SwarmCache cost less than 20 KB (bootstrap costs excluded). This was not the case for all swarms. We recorded 4 refreshes that required more than 100 KB of network traffic. These incidental high costs are caused by certain clients sending enormous PEX messages.

The variation in network traffic is caused by the degradation of the SwarmCache. The second plot in Figure 5.3 shows the size of the SwarmCache, how many peers are replaced each refresh, and the number of peers received via PEX divided by the SwarmCache size. When a peer no longer responds, more connections are made and more network traffic is generated to find a replacement.

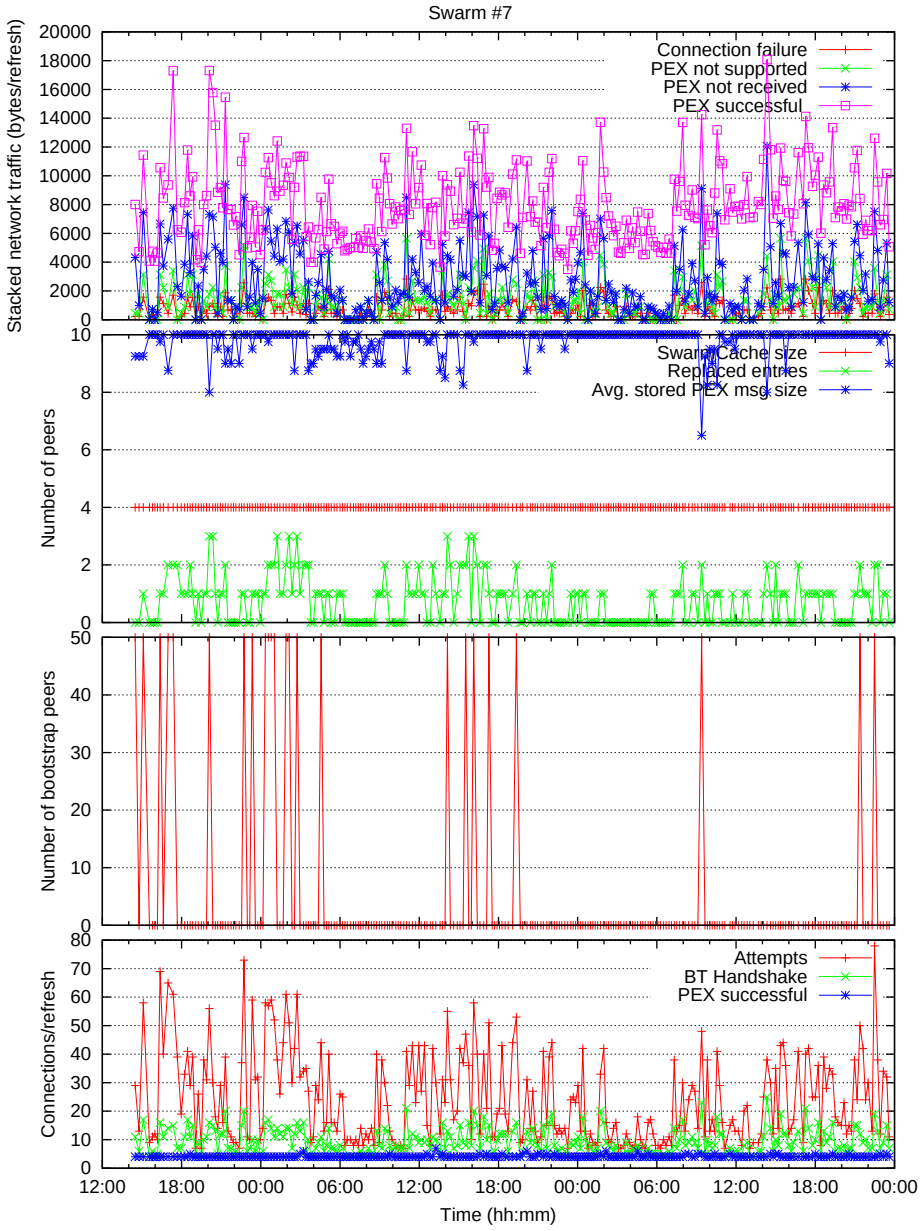


Figure 5.3: Performance of successive RePEX refreshes every 10 minutes.

In the third plot of Figure 5.3 we can see how often the RePEX algorithm has to fall back to a bootstrap mechanism –which is in this case the central tracker– and how many peers it receives. Since the experiment started with empty SwarmCaches for each swarm, the first bootstrap is always required. The subsequent tracker calls, however, are caused when a replacement needs to be found, but none of the peers received via PEX earlier are responsive. Table 5.1 shows the number of times a bootstrap was needed for all the swarms. In all instances, the tracker fallback rate is less than once per hour. Since a tracker call is nothing more than a simple HTTP GET request [36], these costs are negligible. When a tracker does not support compact peer lists, it needs about 73 bytes per peer [32]. Since a tracker response usually contains 50 peer addresses, 4 KB would be a safe upper bound for the costs of a tracker call.

Swarm	#Refreshes	#Bootstraps	Bootstrap rate $(1/h)$
#1	266	13	0.23
#2	266	12	0.21
#3	266	38	0.66
#4	266	50	0.87
#5	266	23	0.40
#6	265	27	0.47
#7	265	24	0.42
#8	265	6	0.10
#9	265	7	0.12
#10	266	22	0.38
Ubuntu	265	47	0.82

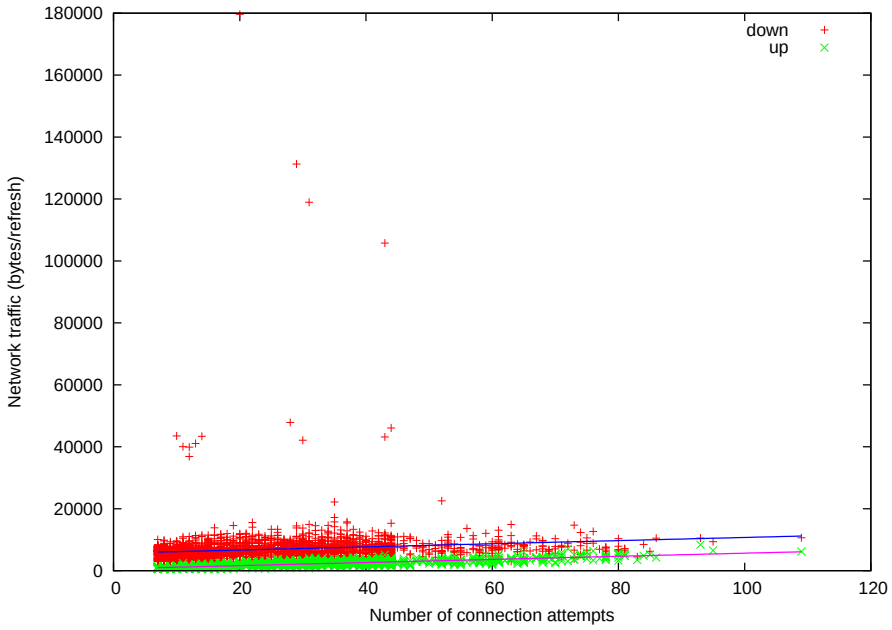
Table 5.1: The number of refreshes and bootstraps per swarm.

The number of connection attemps, connections made and the amount of generated network traffic are obviously correlated. Figure 5.4 shows that they are approximately linearly related. Furthermore, 95% of the refreshes require 13 KB or less. On average, a refresh costs 8.5 KB.

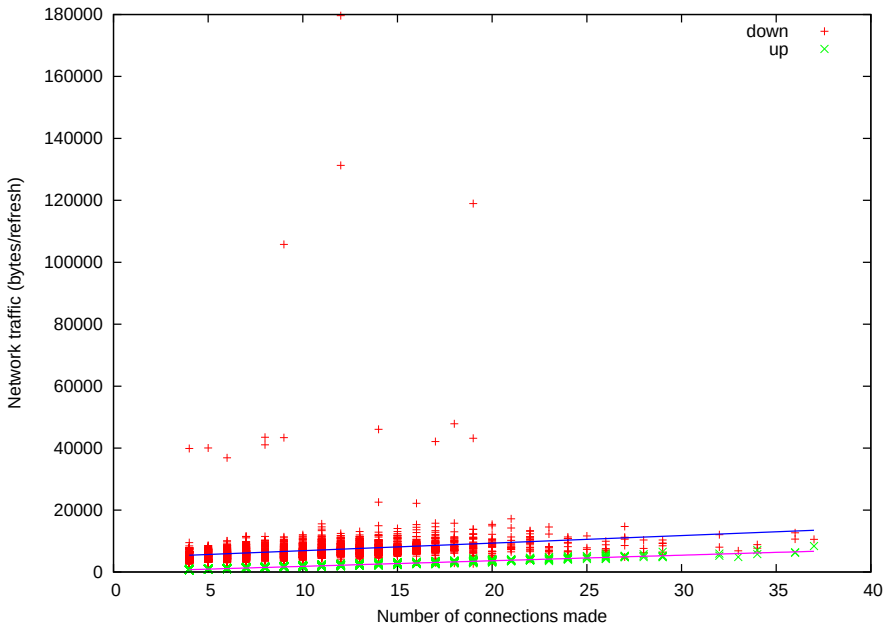
Quality of the SwarmCache

To determine the quality of the SwarmCache, we need to consider the scenario where some Tribler peer queries our SwarmCache via the extended keyword search. The quality of the SwarmCache is then the fraction of responsive peers. A query can arrive between any two refreshes, at time $t_i < t < t_{i+1}$. Let K_{t_i} denote the SwarmCache at time t. We assume that all peers replaced at time t_{i+1} are not responsive during the interval. The quality of a SwarmCache K_{t_i} then is

$$Q(K_{t_i}) = 1 - \frac{\left|K_{t_i} \setminus K_{t_{i+1}}\right|}{|K_{t_i}|}.$$



(a)



(b)

Figure 5.4: Network traffic versus the number of (a) connection attempts and (b) connections made.

Note that the interval between two refreshes is not necessarily constant as refreshes are scheduled sequentially by the scheduler, not concurrently. Therefore, when computing the distribution of the SwarmCache quality for a specific swarm, we use the length of the intervals as the weight for each Q. The quartiles of the distribution of Q for each swarm are shown in Figure 5.5. We can see that at least in 75% of the time at least half of the cache is responsive for swarms #5, #6 and #8, and at least 75% of the cache for the other swarms.

The box plot also shows that the quality of the SwarmCache sometimes drops to 0. This means that if someone would query the cache at these times, he would not be able to use the contents of this cache to join the swarm. Table 5.2 lists the number of intervals in which the quality dropped to 0.

Swarm	#Refreshes	#Intervals where $Q = 0$
#3	266	1
#4	266	1
#5	266	2
#6	265	5
#10	266	1

Table 5.2: The number of intervals in which the quality of the SwarmCache was 0 for popular swarms.

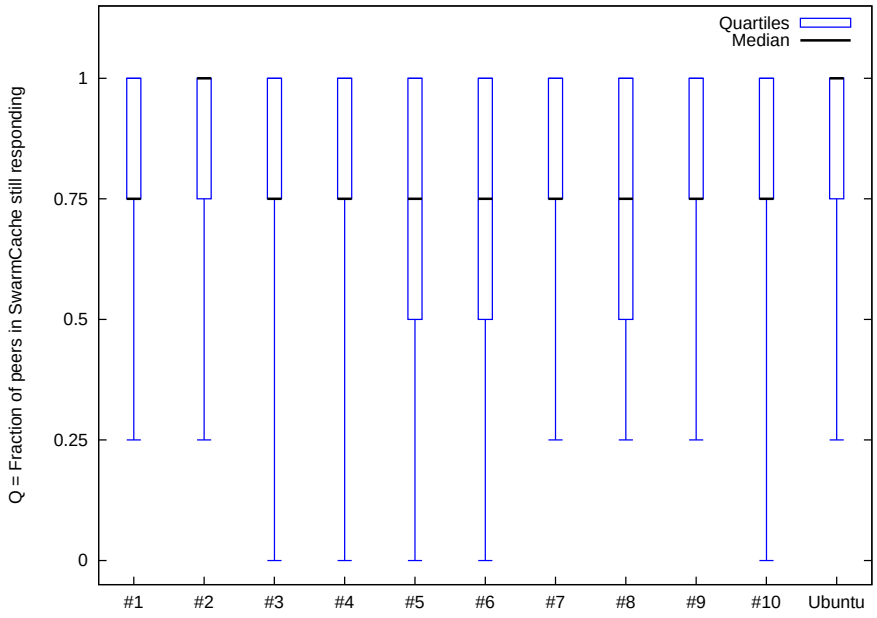


Figure 5.5: SwarmCache quality for popular swarms.

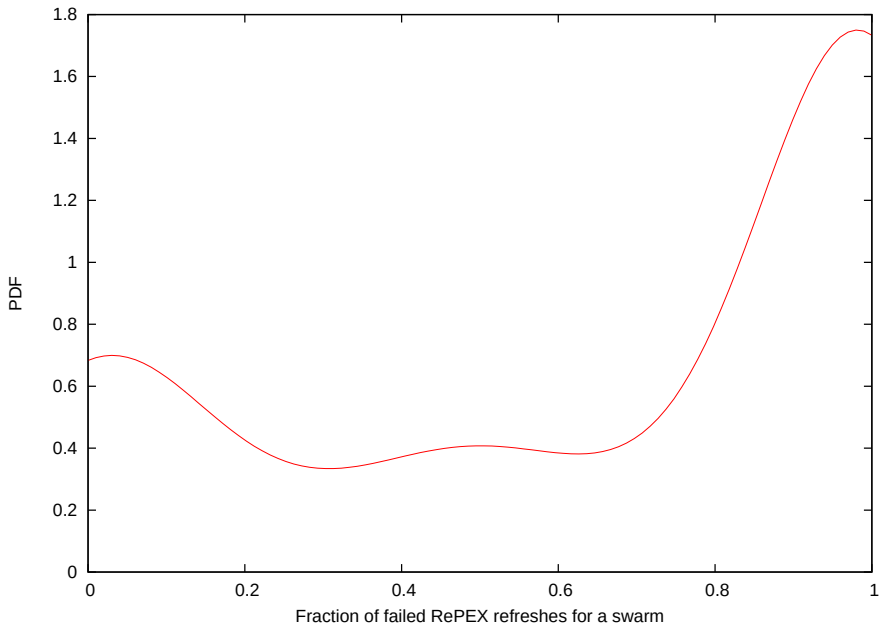


Figure 5.6: Probability density of the fraction of failed RePEX refreshes for a swarm.

5.3 Deployment

We have deployed the current implementation of our RePEX algorithm with the beta version of Tribler 5.2. The configuration was similar to the setup used in Section 5.2, except SwarmCaches were refreshed every 20 minutes instead of every 10 minutes.[1] Additionally, the implementation recorded performance data to a log file. From March 9th to March 30th, 2010, we crawled the Tribler overlay network to retrieve these logs. During this crawl, we have encountered 149 Tribler PermIDs [41] with the deployed algorithm, but only 34 of them actually performed the RePEX algorithm on swarms. In total, 200 unique swarms were tracked, of which two swarms were each tracked by two Tribler peers. Seven of the 34 Tribler peers tracked 10 or more swarms. For half of the tracked swarms, the SwarmCache was refreshed at most 9 times.

A first look at the collected performance data shows unexpected results. Many of the SwarmCache refreshes failed and resulted in an empty SwarmCache. Computing the fraction of failed refreshes for each swarm, i.e. the quotient of the number of failed refreshes and the total number of refreshes for a swarm, reveals two peaks in its distribution (Figure 5.6). For a small portion of swarms tracked by the Tribler beta users, the RePEX algorithm manages to find peers for in the SwarmCache most of the time, but for the majority of the tracked swarms, it seems to fail nearly

[1] The SwarmCache refresh rate parameter was not set to 10 minutes for the beta release of 5.2 due to a forgotten commit.

constantly.

To investigate this matter, we have picked a swarm with a high failure ratio. The peer tracking this particular swarm attempted to refresh the SwarmCache 306 times. Of the 306 attempts, 294 failed. For each failed refresh, however, connection attempts to swarm peers were made. In 276 of the failed refreshes, between 1 and 28 BitTorrent connections were actually established, but not a single useful PEX message was received. For each of the 18 remaining failed attempts, 6 peers were tried in vain. We can see that this was a rather small swarm. Since no PEX messages were received, we conjecture that, under the assumption each swarm peer actually supported PEX, the swarm mainly consisted of seeders. Seeders are normally not connected to each other, and hence, in a seeders-only swarm, seeders do not have any neighbours. That means that a PEX message from a seeder in a seeders-only swarm would be empty, if sent at all. Since our current implementation does not consider empty PEX messages, none of the seeders in a seeders-only swarm would be included in the SwarmCache.

Our algorithm also performed worse in the wild when it comes to bootstrapping. For 192 SwarmCaches, a central tracker or the DHT had to be contacted in more than 90% of the refresh attempts. Part of this is due to seeders-only swarms we just discussed and part of this is due to the larger refresh interval. However, we also observed that the time between two refreshes was sometimes much longer than the configured refresh interval. The much longer intervals were simply caused by users logging off and restarting Tribler at a later time. One extreme example can be seen in Figure 5.7 where a user was offline for two days.

Unsurprisingly, in cases where our RePEX algorithm did not constantly fail, we observed that the larger refresh interval reduced the quality of the SwarmCaches. Figure 5.8 shows the SwarmCache quality for swarms tracked by a particular Tribler client. We will ignore the swarms with a median quality of 0 as these are instances with a high fail ratio. We see that the lower quartile even hits 0 for swarms #4, #5, #13, and #15. For these swarms, there were a significant number of intervals where the SwarmCache quality was 0 as shown in Table 5.3, with the exception of swarm #5. For swarm #5, the quality distribution is skewed because the Tribler peer apparently went offline for 7 hours and 47 minutes between two refreshes. For swarms #3, #7, and #14, the RePEX algorithm did relatively well.

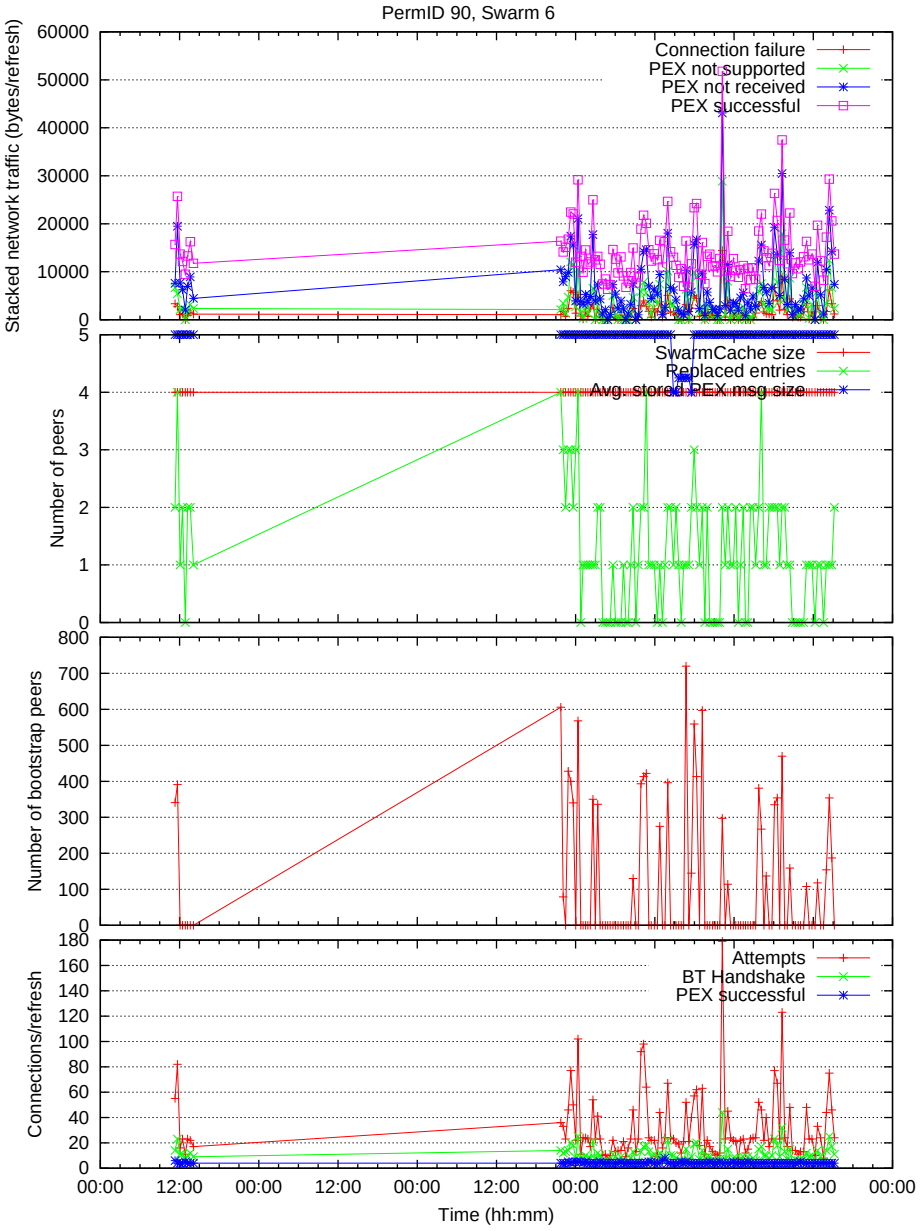


Figure 5.7: Performance of RePEX with a refresh interval of 20 minutes.

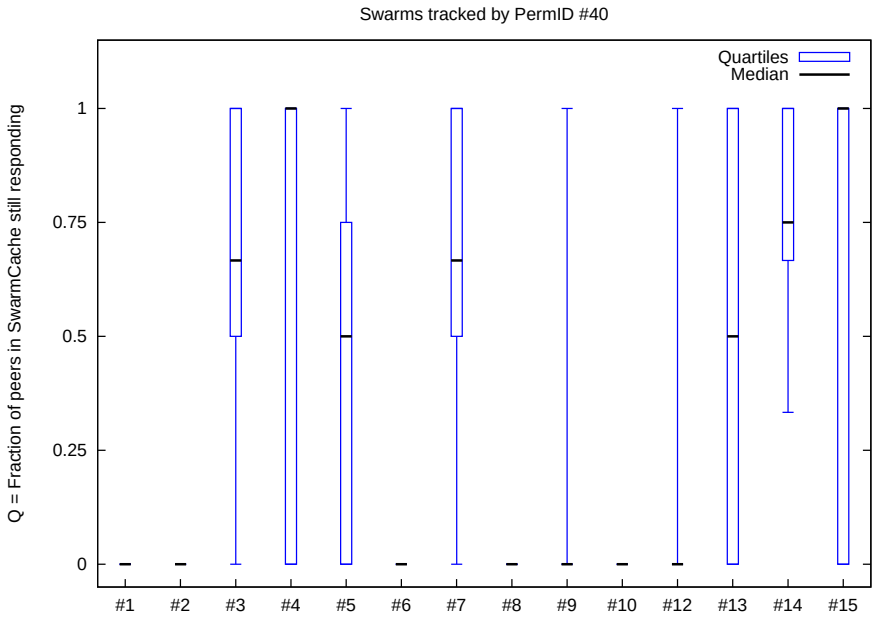


Figure 5.8: SwarmCache quality for swarms tracked by PermID #40.

Swarm	#Refreshes	#Intervals where $Q = 0$
#3	73	8
#4	28	9
#5	71	4
#7	30	2
#13	27	12
#14	62	0
#15	36	18

Table 5.3: The number of intervals in which the quality of the SwarmCache was 0 for swarms tracked by PermID #40.

5.4 Evaluation

We have seen that 95% of the time the cost of our RePEX algorithm is 13 KB or less per refresh. For a refresh interval of 10 minutes and the tracking of 50 swarms, the RePEX algorithm costs 65 KB/min or 1.1 KB/s. Compared to running a Kademlia DHT node, this cost is rather high. Running a Kademlia DHT node only costs 12.10 KB/min [32]. However, 1.1 KB/s is still within a reasonable range for today's broadband connections.

The challenge is to tweak the configuration parameters, such that we can decrease the costs, yet increase the quality of refreshed SwarmCaches, and reduce the number of times the algorithm relies on a bootstrapping mechanism, i.e. currently, how many times central tracker is called. Using a simple binomial probability model, we can model the probability that the quality of a SwarmCache becomes 0 as follows:

$$P(Q = 0) = (1 - p(t))^{S_1},$$

where $p(t)$ is the probability that a peer is still online after t minutes. We will use Table 3.2 to estimate $p(t)$. Increasing the refresh interval from 10 minutes to 20 minutes, would increase $P(Q = 0)$ with a factor of

$$\frac{P(Q = 0 \mid S_1 = 4, p(20\,\text{min}) = 0.800)}{P(Q = 0 \mid S_1 = 4, p(10\,\text{min}) = 0.857)} = 3.83.$$

If we do not want this probability to grow when increasing the refresh interval to 20 minutes, then we need to store $S_1 \geq 7$ peers in the SwarmCache instead. This, however, would also mean we need to make more successful connections each refresh, which would increase the bandwidth cost. Assuming linear growth, the gain would be minimal:

$$100\% \cdot (1 - \frac{10\text{min}}{20\text{min}} \cdot \frac{7\text{peers}}{4\text{peers}}) = 12.5\%.$$

If we want to effectively reduce the number of times the algorithm has to rely on the central tracker, we must know the effect of the chosen parameters. We can model the chance that the tracker is contacted as follows:

$$P(\text{bootstrap}) = b(S_1, S_2) =$$

$$\sum_{i=0}^{S_1} P(X = i \mid X \sim \text{Bin}(S_1, p_1)) \cdot P(Y \leq S_1 - i - 1 \mid Y \sim \text{Bin}(S_1 \cdot S_2, p_2)),$$

where $p_1 = p(10\text{min}) = 0.857$ and $p_2 = 0.14p_1$. This model assumes the SwarmCache contains S_1 primary peers and for each primary peer S_2 secondary peers. The tracker is contacted when not enough secondary peers are alive to replace the non-responding primary peers. The factor 0.14 in p_2 represents the useful fraction of a PEX message.

Using this model, we can compute the value of b for our previously chosen values of S_1 and S_2:

$$b(4, 5) = 6.00\%$$

Comparing this with Table 5.1, we notice it underestimates the probability. This can be attributed to the fact that this model assumes each SwarmCache refresh is independent. Nevertheless, we can use this model to predict the effects of changing the parameters S_1 and S_2.

Doubling S_1 results in a $b(2S_1, S2) = 2.62\%$. These are slightly better odds, but as previously argued, this will cost more bandwidth. Doubling S_2 on the other hand results in $b(S_1, 2S_2) = 0.71\%$. This is a much better improvement, but at the expense of disk storage. Fortunately, disk space is much less scarce than bandwidth.

A more effective way to reduce the chance of contacting the tracker would be to improve the chance a secondary peer is reachable and supports PEX. However, it is hard to improve $p_2 = 0.14p_1$. Improving the factor 0.14 would mean we would have to improve existing PEX implementations and deploy them. Improving p_1 means we would have to incent peers to stay online longer. Instead, we can improve the algorithm by not relying on old secondary peers in the current SwarmCache, but by preferring fresh secondary peers we may already know for the refreshed SwarmCache. Assuming a PEX message contains on average 50 peers, our new function b' becomes:

$$\begin{aligned} b'(S_1, S_2) = &\sum_{i=0}^{S_1} P(X = i \mid X \sim \text{Bin}(S_1, p_1)) \\ &\sum_{j=0}^{50i} P(Z = j \mid Z \sim \text{Bin}(50i, 0.14)) \cdot \\ &\quad P(Y \leq S_1 - i - j - 1 \mid Y \sim \text{Bin}(S_1 \cdot S_2, p_2)). \end{aligned}$$

The new model predicts this improvement would be quite effective, as the chance that the algorithm has to bootstrap drops below one percent:

$$b'(4, 5) = 0.04\%$$

We have also seen that our RePEX algorithm has some shortcomings. In small and seeders-only swarms it fails to create an updated SwarmCache. Only considering useful PEX messages, i.e. PEX messages with at least one IP, is backfiring on us in this situation. We will have to study small and seeders-only swarms in more detail. We will also have to consider when the RePEX algorithm is allowed to include peers in a SwarmCache that are reachable, but did not send a non-empty PEX message.

Another deficiency of our algorithm is that it does not consider the phenomenon of peers going offline for hours or even days. In such circumstances, it is often pointless trying to check whether the peers in the SwarmCache are still alive.

Instead, it would be better to immediately use e.g. a distributed tracker found via 2-Hop TorrentSmell and use that to bootstrap the SwarmCache.

Chapter 6

Conclusions and Future Work

In this chapter, we give a summary of the problem we have solved and state our conclusions. We then propose future work that can be done to improve our current solution.

6.1 Summary and Conclusions

Swarm discovery is the problem of finding peers that are downloading the same file. The current main solution in BitTorrent is to use a central tracker which tracks all peers in a swarm. The centralized approach of this solution, however, is not scalable at all. Other currently deployed solutions are the DHT and the PEX extension protocol. The DHT does not provide workable scalability and is known to be slow, something that is unacceptable for highly demanding applications such as P2P television. PEX itself was designed to speed up the existing swarm discovery solutions, but was not designed to be used standalone. We have shown that the current PEX implementations are not efficient. Only a small fraction of peers discovered through PEX are actually reachable.

In order to solve the swarm discovery problem more effectively, we have designed 2-Hop TorrentSmell on top of Tribler's keyword search and the PEX protocol. We have implemented the RePEX algorithm that relies on PEX, which is one of the two parts of 2-Hop TorrentSmell. From analysis and measurements, we can state our most important conclusions.

- 2-Hop TorrentSmell is based on a *fast* infrastructure. By leveraging the existing zero-server keyword search mechanism of Tribler featuring a median response time of 325 milliseconds, we expect that finding peers for a swarm would take less than a second.
- 2-Hop TorrentSmell is *scalable*. Popular download swarms give rise to more distributed trackers, over which the load can be distributed.
- 2-Hop TorrentSmell is *effective*. The pull-based design of RePEX checks the connectability of primary peers in the SwarmCache. In large swarms

the median fraction of responsive peers in the SwarmCache is at least 0.75, which is almost twice as high as the median fraction of responsive peers in a tracker response and more than 3 times the median fraction of responsive peers in a PEX message.

- 2-Hop TorrentSmell has a *deployability advantage*. Tracking of swarms is done by leveraging existing protocol features. This means that members of the swarm are not required to implement 2-Hop TorrentSmell. Only a few will suffice as they will track the swarm using the widely deployed PEX protocol.

- 2-Hop TorrentSmell can be made *secure* using a reputation system. Such a system would provide a reputation function that can be used to rank Tribler peers, so we can send queries to peers we perceive as most trustworthy.

We believe that 2-Hop TorrentSmell is a simple, yet effective solution for tracking large BitTorrent swarms and can be tweaked and extended to support smaller swarms as well. By leveraging existing protocols, we can incrementally work our way to a fully, distributed P2P network, devoid of any central components.

6.2 Future Work

The evaluation of our RePEX implementation showed that the current version of 2-Hop TorrentSmell will only work for large swarms. The shortcomings of our algorithm and the possible improvements are as follows:

- The current implementation does not work for small and seeders-only swarms. We need to study the behaviour of BitTorrent clients in these small swarms if we want to effectively track these swarms. We also have to consider a more lenient RePEX algorithm that stores peers that are reachable, but did not send a useful PEX message.

- The current implementation will indiscriminately try to refresh a SwarmCache, regardless of how old it is. Peers that were offline for an extended period of time should not try to refresh an ancient SwarmCache when they come back online. Instead, they should bootstrap right away.

- The scalability of our solution can be improved by introducing new Tribler overlay messages to disseminate information of *current* downloaders. In the current solution, only peers that have finished downloading are known to be tracking. However, there is no reason why a Tribler peer that is currently downloading a file cannot also serve as a distributed tracker.

- The current implementation only uses secondary peers stored in the SwarmCache for backup. We can greatly reduce the number of times the RePEX algorithm needs to bootstrap by preferring new secondary peers found during

a refresh over the older stored secondary peers. The current implementation does not consult newly found secondary peers at all during a refresh.

Addtionally, we have not fully implemented 2-Hop TorrentSmell yet. We should implement the missing extended keyword search mechanism and subject it to extensive measurements. Furthermore, research on distributed reputation systems should be of use to make Tribler's keyword search more secure.

Bibliography

[1] ABC – BitTorrent client.
http://pingpong-abc.sourceforge.net/.

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

[3] ANT Censuses of the Internet Address Space.
http://www.isi.edu/ant/address/.

[4] Azureus (now called Vuze) – BitTorrent client.
http://www.vuze.com/.

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

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

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

[8] BitTorrent.org: DHT Multitracker Metadata Extension, 2008.
http://www.bittorrent.org/beps/bep_0012.html.

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

[10] BitTorrent.org: The BitTorrent Protocol Specification, 2008.
http://www.bittorrent.org/beps/bep_0003.html.

[11] 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.

[12] B. Cohen. Personal announcement: BitTorrent - a new P2P app.
http://finance.groups.yahoo.com/group/decentralization/message/3160.

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

[14] 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.

[15] C. Dale, J. Liu, J. Peters, and B. Li. Evolution and Enhancement of BitTorrent Network Topologies. *Proceedings of IEEE IWQoS*, 2008.

[16] Deluge – BitTorrent client.
http://deluge-torrent.org/.

[17] 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.

[18] 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.

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

[20] P. Garbacki, A. Iosup, J. Doumen, J. Roozenburg, Y. Yuan, M. ten Brinke, L. Musat, F. Zindel, F. van der Werf, M. Meulpolder, J. Taal, and B. Schoon. Tribler Protocol Specification V0.0.2. `http://svn.tribler.org/bt2-design/proto-spec-unified/trunk/proto-spec-current.pdf`, 2009.

[21] Pawel Garbacki, Alexandru Iosup, Dick H. J. Epema, and Maarten van Steen. 2Fast: Collaborative Downloads in P2P Networks. In Alberto Montresor, Adam Wierzbicki, and Nahid Shahmehri, editors, *Peer-to-Peer Computing*, pages 23–30. IEEE Computer Society, 2006.

[22] 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.

[23] libtorrent – C++ BitTorrent library. `http://www.rasterbar.com/products/libtorrent/`.

[24] 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.

[25] 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.

[26] M. Meulpolder, J. A. Pouwelse, D. H. J. Epema, and H. J. Sips. BarterCast: A practical approach to prevent lazy freeriding in P2P networks. 2009.

[27] 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. In *2009 11th IEEE International Symposium on Multimedia*, pages 342–349. IEEE, 2009.

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

[29] P2P-Next. `http://www.p2p-next.org/`.

[30] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. J. Epema, M. Reinders, M. R. van Steen, and H. J. Sips. Tribler: A social-based Peer-to-Peer system. *Concurr. Comput. : Pract. Exper.*, 20(2):127–138, 2008.

[31] 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.

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

[33] T. Silverston and O. Fourmaux. P2P IPTV measurement: a comparison study. *Arxiv preprint cs/0610133*, 2006.

[34] 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.

[35] 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.

[36] TheoryOrg: BitTorrent Protocol Specification v1.0. `http://wiki.theory.org/BitTorrentSpecification`.

[37] Transmission: Extended messaging. http://trac.transmissionbt.com/browser/trunk/doc/extended-messaging.txt.
[38] Tribler – BitTorrent client. http://www.tribler.org.
[39] Tribler Wiki: 4th Generation of P2P. https://www.tribler.org/trac/wiki/4thGenerationP2P.
[40] Tribler Wiki: Content Search. https://www.tribler.org/trac/wiki/ContentSearch.
[41] Tribler Wiki: Secure, Permanent Peer Identifiers. http://www.tribler.org/trac/wiki/PermID.
[42] µTorrent – BitTorrent client. http://www.utorrent.com/.
[43] J. Zhang, L. Liu, L. Ramaswamy, and C. Pu. PeerCast: Churn-resilient end system multicast on heterogeneous overlay networks. *Journal of Network and Computer Applications*, 31(4):821–850, 2008.

Appendix A

RePEX Specifications

This document presents the specifications of the RePEX module for Tribler. In Section A.1 we give an overview of the RePEX source code. In Section A.2, we give an overview of the original architecture of Tribler's download engine. In Section A.3, we describe the integration of the RePEX module and the interaction with the existing components. In Section A.4, we explain the storage of the SwarmCache data structure and its API. In Section A.5, we list the configuration parameters of the RePEX module.

A.1 RePEX source code

SVN: `http://svn.tribler.org/abc/branches/mainbranch/`
Revision: 32289

Tribler/Core/DecentralizedTracking/repex.py The main file of RePEX. This file contains:

- Configuration parameters (Section A.5).
- RePEXerInterface interface – Defines the public methods that Tribler's download engine can call.
- RePEXer class – Implements RePEXerInterface. Responsible for running the RePEX algorithm, i.e. updating the SwarmCache for a Download. For each Download in RePEX state, an instance is made.
- RePEXerStatusCallback interface – Defines the required methods to observe RePEXer instances.
- RePEXScheduler class – Implements RePEXerStatusCallback and is a singleton. Responsible for starting Downloads in RePEX state and observes all RePEXer instances.

- RePEXLogger class – Implements RePEXerStatusCallback and is a singleton. Logs information about SwarmCache refreshes for measurement purposes.
- RePEXLogDB class – Singleton class. Contains logs on SwarmCache refreshes.

Tribler/Core/Statistics/RepexCrawler.py Crawler used in Section 5.3 to retrieve logs on SwarmCache refreshes.

Modified files:

Tribler/Core/API.py Documents RePEX additions to the Tribler API.

Tribler/Core/simpledefs.py Introduces DLSTATUS_REPEXING as a new Download status.

Tribler/Core/DownloadState.py Modified to handle the new Download status. Provides a new public API method called get_swarmcache().

Tribler/Core/Session.py Downloads can now be created and started with some initial download status.

Tribler/Core/APIImplementation/DownloadImpl.py Propagates initial download status to SingleDownload. Responsible for loading and storing a SwarmCache from and to the Download's persistent state (pstate).

Tribler/Core/APIImplementation/SingleDownload.py Responsible for instantiating and hooking up a RePEXer when started in DLSTATUS_REPEXING. Is now also notified when a Download is restarted.

Tribler/Core/BitTornado/download_bt1.py Refactored the startRerequester() method into createRerequester() and startRerequester().

Tribler/Core/BitTornado/SocketHandler.py Contains network traffic counters for measurement purposes.

Tribler/Core/BitTornado/Encrypter.py Informs a RePEXer instance on events triggered by a BitTorrent connection.

Tribler/Core/BitTornado/Connecter.py Informs a RePEXer instance on events triggered by a BitTorrent connection. Counts received PEX messages.

Tribler/Core/BitTornado/DownloaderFeedback.py Exposes the PEX message counter for each peer in the neighbourhood set.

Tribler/Core/BitTornado/Choker.py The connection_lost() method was made more robust to work with the RePEX code.

Tribler/Core/BitTornado/MessageID.py Defines message IDs for the crawler used in Section 5.3 to retrieve logs on SwarmCache refreshes.

Tribler/Core/Overlay/OverlayApps.py Registers the crawler used in Section 5.3 to retrieve logs on SwarmCache refreshes.

Tribler/Main/tribler.py Starts the RePEXScheduler and RePEXLogger.

Tribler/Main/vwxGUI/LibraryItemPanel.py Handles the new Download status.

Tribler/Policies/RateManager.py Handles the new Download status.

A.2 Original architecture of Tribler's download engine

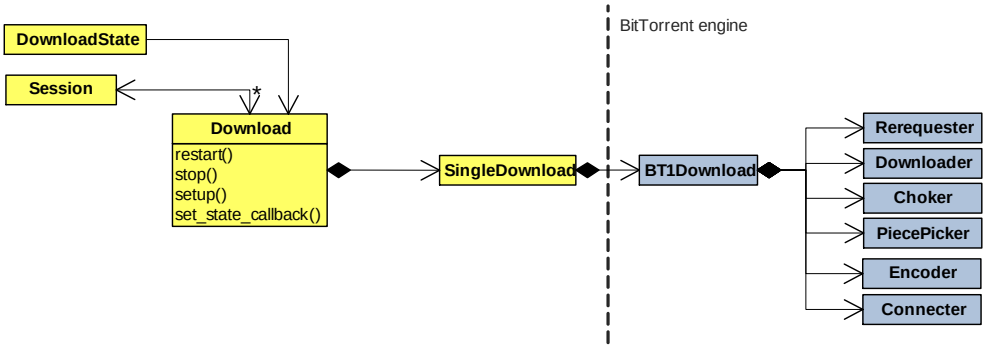


Figure A.1: Original architecture.

A simplified version of Tribler's architecture containing the relevant classes prior to the addition of the RePEX module is shown in Figure A.1. In Tribler, a download is represented by an instance of the Download class. All Downloads are managed by the Session singleton. The state of a Download is represented by an instance of the DownloadState class. A Download does not own an instance of DownloadState, but it instead creates a new instance everytime its state is queried through the set_state_callback() method.

When a Download is created or restarted, an instance of SingleDownload is created, which represents the internal classes of the BitTorrent engine. SingleDownload creates an instance of the BitTorrent engine by instantiating BT1Download and then calling BT1Download.startRerequester(). This method creates an instance of Rerequester (the only class BT1Download does not instantiate on its own) and starts it. The Rerequester is responsible for periodically contacting the tracker.

When it receives a peer list from the tracker, it sends it to the Encoder by calling Encoder.start_connections().

The other classes of the BitTorrent engine are notified when a successful connection has been made, such that e.g. PiecePicker can start sending piece requests. For more details on the BitTorrent engine, please consult "How BitTornado Works" by Arno Bakker.[1]

A.3 Integration of the RePEX module

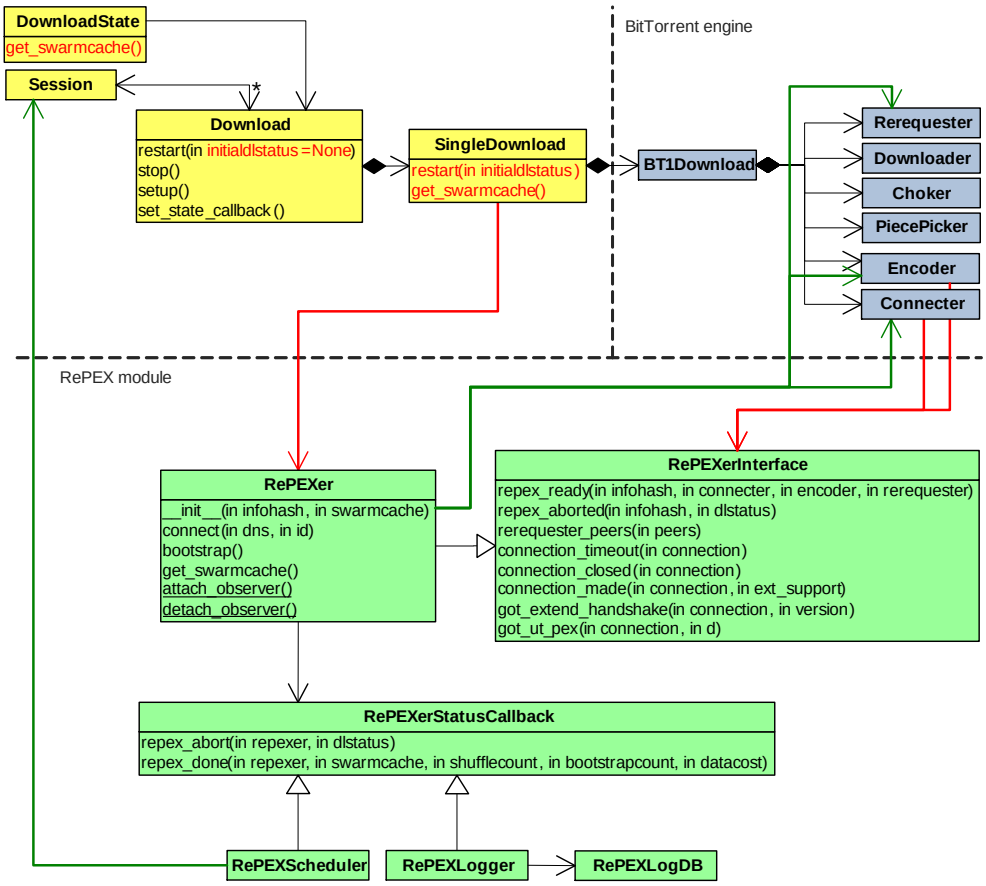


Figure A.2: Integration of the RePEX module.

Figure A.2 depicts the integration of the RePEX module. In order to run the RePEX algorithm for a particular swarm, the RePEXer class needs to be able to start connections, get notified on connection events, and be able to bootstrap. For this it needs an instance of the Encoder, Connecter and Rerequester classes. Single-

[1] http://www.tribler.org/trac/attachment/wiki/TriblerArchitecture/BitTornado-operation-20050916.pdf?format=raw

Download provides these instances by calling the RePEXer's repex_ready() method. SingleDownload is also responsible for creating a RePEXer instance when the Download is started in the RePEX state.

A Download can be started in the RePEX state by calling the restart() method on a Download in the *stopped* state with the initialdlstatus argument set to DLSTATUS_REPEXING. Since the Download was not running, it creates a new instance of SingleDownload and passes the initialdlstatus through the download configuration dictionary. Prior to instantiating the BitTorrent engine, SingleDownload creates an instance of the RePEXer class. It continues with hooking the RePEXer into the BitTorrent engine by pausing the engine, creating the Rerequester without starting it, and adding the RePEXer to the Encoder and Connecter (see SingleDownload.hook_repexer()). Finally, it informs the RePEXer it is ready to start the RePEX algorithm by calling repex_ready().

Originally Download's restart() method was a no-op for a running download. Now that a Download can be in an additional state, this is no longer the case. A call to Download.restart() is now always delegated to SingleDownload. If the RePEX algorithm is running and initialdlstatus is not set to DLSTATUS_REPEXING, SingleDownload will interpret this command as a request to abort the RePEX algorithm and to continue normally, i.e. seeding the swarm. When requested to stop RePEXing or when being shutdown, SingleDownload unhooks the RePEXer (see SingleDownload.unhook_repexer()) and informs the RePEXer it has to abort by calling its repex_aborted() method.

Stopped Downloads are periodically scheduled to refresh their SwarmCache. The RePEXScheduler achieves this by periodically requesting a list of DownloadStates from Session using Session's set_download_states_callback() method and scanning for a Download in the stopped state with an outdated SwarmCache. When it has found a suitable Download, it stops requesting a list of DownloadStates from the Session and starts the Download in the RePEX state. When a Download is done RePEXing or is aborted, the scheduler is notified via the RePEXerStatusCallback interface and continues to periodically request and check a list of DownloadStates.

To receive notifications from all RePEXer instances, the RePEXScheduler registers itself on startup using RePEXer's attach_observer class method. The RePEXScheduler is instantiated and started in ABCApp.startAPI(), which is located in Tribler's main script, tribler.py. In the same location, the RePEXLogger gets started.

A.4 The SwarmCache data structure

The SwarmCache data structure is discussed in Section 5.1 and shown in Figure 5.2. The SwarmCache is stored in the persistent state (pstate) of a Download. This persistent state is automatically loaded and stored by Tribler's Session and LaunchManyCore classes.

A Download's SwarmCache is accessible through its DownloadState. Depen-

ding on the Download's status, DownloadState can return one of the following:

- If the Download is running, i.e. downloading or seeding, a sample of the current neighbourhood is returned. This sample only includes local and PEX supporting peers.
- If the Download is not running, i.e. its status is stopped, the last stored SwarmCache in the Download's pstate is returned. If there is none, an empty SwarmCache is returned.

A Download will store a SwarmCache to its pstate when it is being stopped. If it was running the RePEX algorithm, it will query the associated RePEXer instance (via SingleDownload) for an updated SwarmCache and store the result. Otherwise, it will query its DownloadState for a possibly updated SwarmCache instead.

A.5 Configuration 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 of this thesis.

`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 of this thesis. The sampling currently 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.