Distributed algorithms for improving BitTorrent performance

ANIL CAN AKAY

KTH Electrical Engineering

Master's Degree Project
Stockholm, Sweden

XR-EE-LCN 2010:010

Distributed Algorithms for Improving BitTorrent Performance

ANIL CAN AKAY

Supervisor : György Dàn

Examiner : Viktoria Fodor

Master of Science Thesis
Stockholm, Sweden 2010

Abstract

Among the peer-to-peer systems, BitTorrent has attracted significant attention in research community because of its efficiency, scalability and robustness. BitTorrent utilizes peer contribution to distribute content by splitting the content into many pieces which can be transferred among peers. Unfortunately BitTorrent depends on trackers in order to let peers interested in same content discover each other. Trackers can be considered as a single point of failure and a bottleneck in terms of system scalability. The scalability and availability of the tracker can be improved by introducing multiple trackers, an extension that allows the co-existence of multiple swarms sharing the same content. Existence of multiple swarms that are not aware of each other may cause efficiency to degrade due to piece and bandwidth unavailability in small swarms. Swarm management algorithms therefore aim to increase the swarm sizes virtually at a low cost, consequently increasing piece availability and peer contribution for performance improvement.

In this thesis we developed a framework for measuring the performance of swarm management algorithms in an experimental testbed. The testbed offers the opportunity to perform controlled experiments in different scenarios. An improved PEX protocol was also developed that takes swarm membership information of peers into account to utilize mixing among swarms.

We modified an existing BitTorrent client to implement two swarm management algorithms, Random Peer Migration (RPM) and Random Multi Tracking (RMT) that introduce peers in different swarms to each other by leveraging the Peer Exchange (PEX) protocol. RPM achieves mixing through peers migrating between swarms. RMT allows some fraction of peers to associate with more than one tracker and mix peer information between swarms.

We evaluated the performance of the two swarm management algorithms in torrents in which all swarms are in the steady state and have a publisher always available. The algorithms are estimated to improve the protocol performance around 5% in most scenarios whereas gain around 40% can be observed for small torrents. The algorithms are shown to improve BitTorrent performance without sacrificing the robustness and load balancing properties of the multi-tracker extension.

Keywords: BitTorrent, Dynamic Swarm Management, Multi-Tracker Extension, Peer Exchange (PEX), performance, swarm splitting, distributed algorithms

Acknowledgments

I owe my deepest gratitude to my supervisor, György Dàn. I would like to thank him for offering me the chance to work with a great scientist. This thesis would have never been finished without his continuous encouragement, invaluable advice and guidance.

I also would like to thank to Ognjen for his endless patience and helps.

I would like to thank to my friends, for their great support and friendship. I owe so much to my childhood friends Erkin, Eren, Kuzu and Can for always being with me regardless of the distances. You are part of me.

I am grateful to my friends from college Aykut, Berk, Ali Galip, Deniz, Alev, Gorkem, Okan, Merve, Kazim, Yunus, Cagri, Cigdem and Onur for the invaluable memories in ODTU, Ankara. I always feel you near with me.

I would like to thank my friends Haluk and Bener for their friendship and support during my thesis.

I am indebted to my friends in Sweden Nazim, Mehmet, Oguzhan, Bahadir, Rafet, Erdal, Derya, Ozden and Akif for making me feel at home and thanks to many others which is hard to list all here.

I wish to give special thanks to my brother Onur, for enjoying the life together with me.

The greatest thanks are to my parents, for their support throughout my whole life.

Contents

1. INTRODUCTION ... 1
1.1. Problem Statement ... 1
1.2. Methodology ... 2
1.3. Organization of the Report ... 2
2. BACKGROUND ... 3
2.1. BitTorrent Protocol ... 3
2.1.1. General Description of BitTorrent ... 3
2.1.2. BitTorrent Components ... 4
2.1.3. Protocol Details ... 5
2.1.4. Protocol Extensions ... 7
2.2. Kademlia ... 13
2.3 Related Work ... 17
3. DISTRIBUTED ALGORITHMS DESIGN ... 19
3.1. Swarm Management Algorithms ... 19
3.1.1. Baseline ... 20
3.1.2. Picking Biggest Swarm ... 20
3.1.3. Random Peer Migration ... 21
3.1.4. Random Multi Tracking ... 22
3.2. Improved PEX Protocol ... 25
4. FRAMEWORK IMPLEMENTATION ... 30
4.1. Selecting BitTorrent Client ... 30
4.2. Selecting BitTorrent Tracker ... 32
4.3. Experimental Setup ... 32
4.3.1. Running Experiments in Steady State ... 32
4.3.2. Downscaling Experiments ... 36
4.3.3. Providing Global View to Initial Seed ... 37
4.3.4. Determining Warm-up Period and Experiment Length ... 37
4.3.5. Emulating Real Network Characteristics ... 38
4.3.6. Boundaries on System Resources ... 39
4.4. System Components ... 40
4.4.1. Initializing Computers ... 40
4.4.2. Initializing Experiments ... 42
4.4.3. Running Experiments ... 42

4.4.4. Terminating Experiments 44
4.4.5. Processing Collected Data 45
4.4.6. Resetting Experiment 46
4.4.7. Automating Set of Experiments 47
5. EVALUATION 49
5.1. Evaluation Metrics 49
5.2. Default Experiment Parameters 49
5.3. Results 51
5.3.1. Protocol Performance as a Function of the Torrent Size 52
5.3.2. Experiments with 2 Swarms 56
5.3.3. Experiments with 3 Swarms 58
5.3.4. Performance as a Function of the Number of Swarms 62
5.3.5. Discussion 69
6. CONCLUSION and FUTURE WORK 71

1. INTRODUCTION

Among all available peer-to-peer applications for content distribution, BitTorrent has become the most popular by dominating approximately half of the all Internet traffic [2]. This popularity can be greatly attributed to the efficiency, scalability and robustness of the protocol.

The efficiency of the protocol is obtained by splitting the content into many pieces, which once obtained by a peer, can be shared with others while the download continues. The incentive mechanisms of BitTorrent forces peers to contribute to the system by transferring pieces to each other hence allowing the system to scale well with size of the downloading population. In addition to that contribution of every peer increases the resilience of system against peer departures or failures.

1.1. Problem Statement

Although BitTorrent is known for its efficiency, scalability and robustness; trackers are required for the proper functioning of the system. Trackers can be a bottleneck in terms of scalability and robustness. The number of peers downloading the same content can be limited by the capabilities of the tracker. Furthermore upon tracker failures the peers may fail to discover each other hence harming the data exchange [7].

In order to relieve the dependency on trackers, improvements for the protocol such as multiple trackers extension and distributed trackers (DHT) have been proposed. The distributed trackers use the contributing peers to form a distributed hash table for storing peer contact information hence peers can discover each other by querying the DHT. The multi-tracker extension allows use of multiple trackers for the same content so the overall load can be distributed among the trackers. In addition to that the use of multiple trackers increases the resilience against tracker failures.

Although existence of multiple trackers increases the robustness of the system and performs a fair load distribution among trackers, the efficiency of the system is reduced due to partitioning of the overlay as BitTorrent protocol only allows a peer to be associated with one tracker per content to avoid excessive load increase. Typically larger swarms are known to perform better than smaller swarms if the seed to leecher ratios of swarms are equivalent [3]. Therefore coexistence of multiple swarms that are unaware of each other causes the system to perform worse than they were a single swarm.

To prevent the partitioning two dynamic swarm management protocols, Random Peer Migration (RPM) and Random Multi Tracking (RMT) [17], have been proposed that mixes the peer information of different swarms. These algorithms aim to achieve mixing of swarms therefore increasing the size of swarms virtually. As mentioned earlier increasing swarm size increases the piece availability in the system thus improving the overall protocol efficiency.

The dynamic swarm management algorithms consider 3 important criteria:

- Peers belonging to different swarms of same torrent should be able to discover each other.
- The load on trackers should be balanced.
- The communication overhead due to swarm management should be kept low.

The algorithms mentioned only require modification on peer behaviour hence no additional information maintenance is required by trackers. In Random Peer Migration (RPM) a small fraction of the peers migrate between swarms and in Random Multi Tracking (RMT) some peers are allowed to register to more than one tracker. The mentioned peers mix the peer information of different swarms by leveraging the peer exchange (PEX) protocol. The PEX protocol is also modified to take swarm membership of peers into account during exchange to increase the mixing between swarms. Briefly the aim of the described algorithms is to increase BitTorrent performance through mixing peer information of disjoint swarms without causing an excessive increase in communication overhead. It should be noted that all kinds of decentralized peer discovery methods and the algorithms, RPM and RMT, should be disabled if the peer is connected to a private tracker to respect the protocol.

Apart from the mixing algorithms, another algorithm considering the tracker selection policy of peers, Picking Biggest Swarm (PBS) has been examined. This algorithm tries to group all peers in a single swarm for performance improvement.

1.2. Methodology

The progress of this thesis starts with literature study to discover the details of BitTorrent protocol and observe other works that analyze and improve the protocol. Later some algorithms have been considered as promising and worth to experiment.

In order to use for experimentation and development, some of the existing BitTorrent implementations have been examined and a suitable one has been picked. Details of the client selection process can be observed in Section 4.1.

The testbed has been developed with an iterative approach. Firstly, experiments have been performed for testbed verification. Several reasons for inconsistence results have been detected and experiments have been repeated until collecting acceptable results. Additionally the testbed has been modified to work on multiple computers instead of a single machine so that experiments in larger scale could be performed for evaluation purposes.

The algorithms have been implemented incrementally over the default client code. At first the necessary modules that are common in all the algorithms have been developed. Later the clients have been diverged to branches and specific properties have been implemented.

1.3. Organization of the Report

The remainder of this thesis is structured as follows. Section 2 presents general discussion about BitTorrent based on protocol specification, protocol extensions for peer discovery and related works. Section 3 describes the algorithms that improve BitTorrent performance through swarm management. First the swarm management algorithms are discussed in detail. Later the design details of improved peer exchange protocol are given. Section 4 contains implementation details of the distributed testbed for performing BitTorrent experiments. Important properties of experimental setup and functionalities of deployed components are mentioned. Section 5 contains the evaluation results of the developed algorithms. Finally Section 6 concludes the report and discusses the possible future work.

2. BACKGROUND

This chapter provides background information to the readers that are not familiar enough with BitTorrent or Kademlia protocol and discusses the existing related work.

The BitTorrent section firstly introduces the protocol and the components of a BitTorrent system. Then it goes into more detail of the protocol, describing incentives and local policies of the system. As an end it gives information about specific protocol extensions that aim to improve the information availability of the system such as Multi-Tracker extension, Peer-Exchange (PEX) protocol and Distributed Hash Table (DHT) extension. In order to gain better understanding of DHT extension, unfamiliar readers may also refer to Kademlia section. The Kademlia section briefly discusses the Kademlia protocol, the routing algorithm of Kademlia and the maintenance of routing tables. Finally previous works aiming to increase information availability of swarming systems with various methods are mentioned.

2.1. BitTorrent Protocol

BitTorrent is a peer-to-peer content distribution protocol that gained much popularity in recent years. Recent measurements show that it dominates the Internet traffic by accounting approximately 27-55% of all traffic by February 2009 [2].

2.1.1. General Description of BitTorrent

The main goal of BitTorrent is to efficiently distribute content among users. In a traditional client-server approach, illustrated in *Figure 1*, each client directly retrieves the content from the server causing a load on the server linearly proportional to the number of clients. The scalable approach of BitTorrent aims to relieve the burden of a centralized server by trying to utilize uplink bandwidths of participating clients.

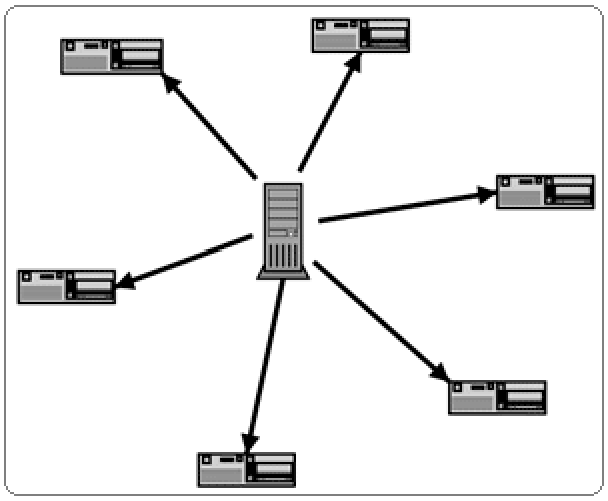


Figure 1: Content distribution via centralized server (Figure from [1])

In order to efficiently benefit from upload capacities of peers the content in BitTorrent is split into many small pieces. The peers can cooperatively exchange pieces among each other so the system can handle large number of peers sharing the same content by avoiding a server bottleneck. The user cooperation in content distribution can be observed in *Figure 2* where peers retrieve pieces from each other and the load on server is relieved by user contributions.

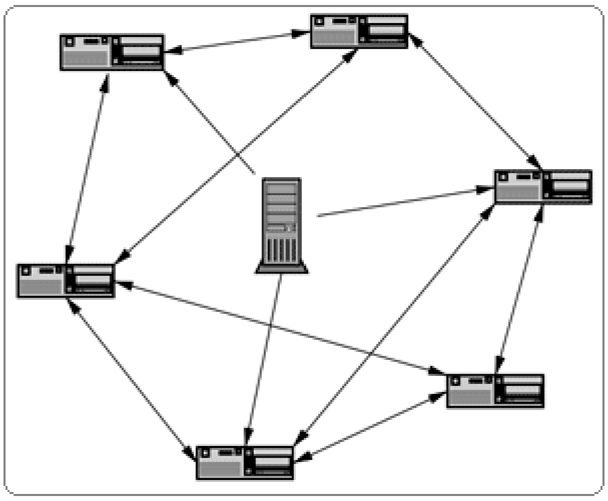


Figure 2: Content Distribution with BitTorrent (Figure from [1])

The peers that are interested in the same content are named as *swarms*. Peers that share the whole content either altruistically or through some incentives are named as *seeds*. The peers that have no pieces or some fraction of them are called *leechers*. Leechers are also expected to contribute piece distribution while receiving pieces from other peers.

In order to join a swarm, a peer initially *announces* itself through a centralized server named *tracker*. The tracker maintains information about peers interested in same content. The peers are identified with their IP address, port number and a peer id. The peer id is a randomly generated string by the peer itself at the start of a new download.

A peer can obtain a partial list of other peers in the swarm from the tracker, that it can connect to and exchange data. The obtained subset forms the basis of the *local neighbourhood* of the peer. In many implementations, the list of peers send by the tracker contains contact information of 50 peers by default. The tracker randomly selects a subset of available peers; resulting with a random graph overlay for the swarm that is known to be popular with its robustness property [6]. Peers can expand their neighbourhood by iteratively querying the tracker within time intervals or through some other mechanisms such as Peer Exchange protocol or Distributed Hash Tables etc. which will be described later.

Every peer in swarm keeps information about the list of pieces it has, namely the *bitmap*. Since peers are provided only a subset of existing peers, they can only gather information about the peers they are in direct contact. The peers exchange their bitmaps in their first contact and notify their neighbours by *HAVE* messages every time they receive a complete piece resulting with a fresh local knowledge.

A peer interested in a specific piece of the content sends a request for download to one of its local contacts that has the piece. The details of data exchange policies among peers will be discussed in the following sections.

2.1.2. BitTorrent Components

A BitTorrent content distribution protocol generally requires the components below:

- An ordinary web server
- A static meta-info file containing information about the torrent
- A BitTorrent tracker
- An original seed that has the whole content
- The end user web browsers

- The end user downloaders

In order to start distributing content using BitTorrent protocol, the peers require a static bootstrapping address to join the swarm and gather information about other peers in the swarm. Generally this process is handled by the centralized component tracker, however other approaches like distributed hash tables are still possible. The serving peer generates a ".torrent" file that contains meta-information about the specific content. Each ".torrent" file is assigned an *info hash* generated using the contents it contains and used to identify the torrents.

The meta-info file contains the address of the web server that the tracker is located and the structure of the content as what file/files are included. As known the protocol divides the content into smaller pieces, generally with size 256-512 KB, which should be carefully selected based on the total amount of the content since too large piece sizes may cause performance degradation and too small pieces cause large ".torrent" files which is required to be stored on web servers. The ".torrent" file contains the piece size and checksum of all pieces that makes it possible for the clients to ensure the correct transmission of pieces. The checksums also protect the system against malicious peers since altered or bogus information can be detected easily.

A BitTorrent download is initiated as the user associates the ".torrent" file to a BitTorrent client. The ".torrent" file can be transferred by any possible method; however the most preferred way is to distribute it through an indexer web site. The BitTorrent client reads the information in the ".torrent" file and announces itself to the given tracker address. The tracker returns a list of peers currently associated with the same content and the content exchange takes place. The newly joining peer is also registered by the tracker to be able to return its address to other peers when requested.

The original publisher that releases the content to distribution should at least send one copy of the total content. However it may leave the system after sending one copy, the content distribution will still continue if the peers having unique pieces do not leave before sending them to others.

2.1.3. Protocol Details

In BitTorrent protocol all logistical problems of data exchange are handled in the interactions between peers. The trackers simply help peers to find each other. In addition to that upload and download amounts are sent to tracker but that is only for collecting statistical information.

As there is no central coordinator for performance improvement, peers are responsible for attempting to maximize their own performance. In order to do so, peers should consider two important points with their knowledge of local neighbourhood. The first point is the *piece selection* strategy of the peers. Peers should decide on what pieces to download next. The second important point is the *selection of the peers to upload to*.

Selecting pieces to download in a good order is a very important task for achieving good performance. A poor piece selection algorithm can end with a situation where the peer only has the pieces that are commonly possessed by other peers or in other words the peer does not have the pieces to trade with others. Poor algorithms may also harm the system's overall performance and availability.

As a first rule, once a peer requests a block of a piece it does not request other pieces until the particular piece is completely received. The policy is named as *strict priority* and it aims to increase the content availability of the system as quick as possible, since a peer can only announce possession of piece after completely receiving it.

As mentioned earlier the peers exchange bitmap information with each other and announce when they completely receive a piece. This information exchange constitutes the local knowledge of a peer about the pieces available in its neighbours. In order to optimize the piece availability, peers decide to download the pieces which the fewest of their neighbours have at first, which is named as *local rarest first* (LRF) policy. Downloading the rarest pieces at first increases the possible number of interested peers in that piece so the piece can be traded with many others. In addition to that, the rarest first policy tries to replicate the rarest pieces in the system as quickly as possible, thus increasing the availability of the content intuitively and increase the system resilience against peer departures. Considering a system with only a single seed, it would be the wisest decision for the seed to send different pieces to different downloaders since redundant downloads would decrease the content availability in the swarm. The rarest first policy again performs well by downloading a large variety of pieces from the seed as peers would be able to determine the pieces that are already available in their neighbourhood and ask the seed for the rare pieces.

Although rarest first policy performs very well by retrieving the rare pieces first and using them in trade for more common pieces, an exceptional case occurs when the peer newly joins the systems or in other words when it has nothing to upload. Since it is important to get a complete piece as soon as possible to trade with other peers, trying to fetch the rarest piece is not a very appropriate decision. As expected, rare pieces are only present in very few peers and many peers are in queue for them so they would be downloaded slower when the peer has nothing to trade. Instead of waiting for the rarest piece, retrieving another piece which is available on many peers would be a better idea since the sub-pieces can be obtained in parallel from different peers. For this reason, peers select the pieces to download at random until the first complete piece is received. This strategy is called *random first piece* and peers switch to rarest first policy after completely downloading the first piece.

Another improvement called *endgame mode* is used in order to avoid potential delays close to a download's finish. It is possible that a piece can be requested from a peer that has slow transfer rates which is not problematic in the middle of a download but can be annoying when it is the final piece. To keep that from happening, once a peer actively sends requests for all the sub-pieces it does not have; it requests all the remaining sub-pieces from all peers it knows. In order to avoid waste of bandwidth due to redundant transfers, cancel messages are used for the obtained blocks. Since the endgame period is a short period, it causes a small amount of bandwidth to be wasted but in turn the end of file is downloaded quickly.

As mentioned earlier the BitTorrent protocol has no central mechanism to control resource allocation, peers are expected to utilize their own bandwidths. In order to do so peers try to download from whoever they can and decide which peers to upload via a variant of *tit-for-tat*. The tit-for-tat policy has the goal to ensure that peers who upload to others are more likely to be able to download. This game theoretic approach encourages peers to contribute to system and avoid the free-riders.

The peers can have connection with multiple peers simultaneously; however they only upload to a fixed number of peers simultaneously which is 4 or 6 by default. The connections that upload is not allowed are called *choked*, meaning temporary refusal of upload. Downloads can still takes place through a choked connection and re-negotiation of the connection is not necessary when the choking period ends or in other words when the peer is *unchoked*.

A downloading peer keeps track of the download rates it receives from its neighbouring peers. Even though there can be different methods to calculate download rates, the base implementation of BitTorrent uses a rolling 20-second average as it is known to perform better than long-term calculations due to the fluctuations caused by joining and leaving peers [5]. The download rates strictly form the basis for deciding which peers to unchoke; a peer only unchokes the fixed number of peers that it receives the best download rates. The choking and unchoking decisions are calculated once every 10 seconds and not applied for the following 10 seconds in order to avoid delays caused by TCP protocol.

The problem with using the tit-for-tat approach alone is that it provides no way to discover other possible fast candidates from the unused connections. In order to solve this issue, BitTorrent protocol uses *optimistic unchoking* algorithm that allows one additional connection to be unchoked at random regardless from the download rates. The optimistic unchoke period by default lasts for 30 seconds which is considered to be enough for the unchoked peer to reciprocate. As the period finishes, the neighbour that has sent the smallest amount of data during the period is choked. Then a new peer is unchoked randomly among the connected peers. Besides helping to discover possible profitable relationships, the optimistic unchokes also gives an opportunity to the newly joined peers to retrieve their first pieces.

Since it is possible for a peer to be choked by all peers it is downloading from, the peers use the *anti-snubbing* strategy. If a peer remains choked by a peer over a minute it assumes that it is *snubbed* by that peer and stops uploading to that peer except an optimistic unchoke case. The peer uses an additional optimistic unchoke to find a better candidate in replacement of the peer snubbed itself.

The tit-for-tat policy does not have applicability to the seeds since they do not have download rates. Instead of download rates, seeds evaluate the peers regarding their upload rates; peers having better upload rates are unchoked. This approach utilizes the upload capacity of the seed and it introduces some level of fairness by preferring peers which does not have utilized their uplinks.

As mentioned earlier each peer is expected to make these decisions based on only the local knowledge they have, the transfer rates between its neighbours. A poor peer selection policy would obviously lead to inefficient distribution of the content which would also decrease the content availability correspondingly.

2.1.4. Protocol Extensions

The BitTorrent protocol with its nature eliminates a single point of traffic congestion with the peer participation in content distribution; however the tracker continues to be a single point of failure as it is the only component that enables coordination between peers. If the tracker managing the swarm becomes unavailable the system will not be able to reachable by new peers and existing peers will not be able to extend their knowledge by querying the tracker. Previous measurement studies point out the shortness of uptime periods of existing trackers as high fraction of them being shorter than a day [8]. Furthermore the failure period of trackers are measured to last around an hour, revealing that most of the failures are caused by software or machine crash rather than a temporary network problem [7].

This section focuses on the current BitTorrent protocol extensions that aim to improve availability. The approaches consider different mechanisms for peers to discover other peers without causing infeasible increase in the load of the system. The first approach considers use of multiple trackers with load balancing purposes or having backup trackers against failures. The second approach introduces use of distributed hash tables (DHT) in order to let the peers work as trackers in a cooperative manner. The third approach makes use of the communication among peers to increase content availability by allowing them to exchange contact information with others. The extensions are named as Multi-Tracker extension, Distributed Trackers (DHT protocol) and Peer Exchange protocol (PEX) respectively and will be discussed in more detail in the following sections.

2.1.4.1. Multi-Tracker Extension

The Multi-Tracker extension allows two or more trackers to track the same torrent instead of only one tracker. Use of multiple trackers originates from two approaches. At first sight it tries to redistribute the load of a single tracker to multiple trackers. Each tracker is expected to know all the peers in the swarm; however each peer chooses a single tracker to announce and query for contact information. The multiple trackers with load balancing purpose exchange information about the peers they know in order to possess a global knowledge of the swarm. Secondly multi-trackers extension can be used for backup purposes. In case of a tracker failure the peers are expected to contact the spare tracker. Multiple trackers for backup purposes are not expected to exchange their knowledge of the swarm. The extension can either be used only for load balancing, backup purposes or both.

The load balancing property of multi-tracker extension can be considered as replicated trackers. The trackers both manage the same swarm; remain synchronized by exchanging contact information but only serve to a portion of the swarm to distribute the load. The availability improvement of this extension actually arises from the backup property. One should also note that multiple trackers for load balancing may also serve for the backup purpose since in case of failures the live trackers will be contacted by the peers. The availability improvement is obvious since it increases the probability of finding at least one available tracker at a time.

In their work [7], the authors observe a correlation among the failures of different trackers or in other words failures of different trackers are not independent. The trackers are observed to be down in similar periods. A possible reason for this correlation can be hosting the trackers in the same machine. The trackers can be considered more vulnerable at the times where the peers cause churns in the system. A flash crowd moving from a failed tracker to another can cause it to fail also and can be a reason for this correlation.

To measure the availability improvement of multi-tracker feature, a *time-aware* method is used. The method keeps history of each tracker and then evaluates the results by checking if at a given time instant there is at least one available tracker.

The multi-tracker feature is shown to increase the availability of the system but the idea behind this improvement does not derive from having a combination of trackers with low availability. In fact the improvement arises from the increased probability of having a highly available tracker in the set of trackers.

The implementation of the extension requires modification to meta-info file, peer and tracker behaviour. Instead of a single announce URL in ".torrent" file, an announce-list section is added to be used by the compatible clients. The announce-list is a list of lists where trackers in the same list are for load balancing purposes and different lists of trackers are the backup trackers. To join a swarm a peer randomly selects a tracker from the first list and keeps using the same tracker until it leaves the system or the tracker fails. The backup lists are used only if all the trackers in the previous list are unreachable. Trackers in the same list, which serve for load balancing, exchange contact information periodically with each other to capture the global view of the swarm but trackers belonging to different lists are not expected to do so.

A potential threat with the multi-tracker feature is that it jeopardizes the connectivity of the overlay; use of multiple trackers can split the swarm into disjoint subsets. A disconnected overlay would certainly harm the content distribution since peers in different subsets will not be aware of each other. This can be caused by tracker failures, churns or long delays between consecutive information exchanges among trackers. A bad implementation of BitTorrent client may also cause disjoint subsets if it does not process the announce-list in correct order as described in the specifications. For instance a peer announcing to backup trackers without testing the trackers in the previous lists would probably get separated from the good swarm. In addition to that if a peer announces to different trackers at each interval instead of sticking to the first tracker it successfully announced; it may cause the trackers to store stale contact information as its departure may not be handled by the correct tracker. Besides, querying all the trackers can cause an excessive increase in load if it is applied by all the peers.

The multi-tracker feature using purely the load balancing property, having a single list with multiple trackers, has been shown not to suffer much from partitioning [7]. Its resistance against partitioning can simply be explained by the periodic information exchanges between trackers. However the backup trackers may cause situations such as on the average a peer in a subset can at most discover half of the peers from available set. A similar situation can emerge with a possible scenario when the first tracker fails and peers start using the backup tracker. As the first tracker recovers back from the failure the newly joining peers will form a different subset disjoint with the peers using backup tracker.

2.1.4.2. Distributed Trackers

The distributed hash table (DHT) extension can directly replace or supplement the central tracker, providing mechanisms for peer discovery and avoiding the single point failure bottleneck of the central tracker. Some of current BitTorrent clients implement a distributed hash table so the clients can form a single DHT infrastructure even though they belong to different swarms. The DHT can be used store/retrieve contact information of peers interested in a specific content. The extension is named as *distributed trackers* since peers cooperatively handle the job of trackers. The info-hash that uniquely identifies each torrent can be used as the *key* and the *values* are the contact information of the peers associated with that content. Peers can perform lookups for the requested content using its info-hash and discover other peers even if they are not in the same swarm. The DHT extension can completely eliminate the tracker dependencies however a hybrid approach certainly performs better.

Current BitTorrent implementations use Kademlia protocol to implement distributed hash tables. As Kademlia stores same information on multiple nodes it provides a high success rate of lookups even in highly dynamic environments. Kademlia with its high resiliency against churn copes well with frequent peer joins and departures in BitTorrent system. As mentioned before a peer should perform lookups with the info-hash of the torrent and receive a set of peer contact information. In order to announce itself to a torrent, a peer should first query the DHT for the closest nodes that store the contact information of the specified torrent. Then the peer simply inserts own contact information to the set of closest nodes. In order to prevent malicious peers associating irrelevant peers with a torrent, a random token should be transferred during this procedure between nodes. For the details of node lookups and storage in Kademlia readers should refer Section 2.2 .

To enable peer joins to DHT, the meta-info file is extended to contain contact information of some nodes that are highly available in the system. The nodes can be manually assigned by the publisher or can be the closest nodes to the info-hash.

In their work [7], the authors measure the effects of DHT extension on information availability. By sending pings to the discovered nodes in DHT, they detect that 70% of the nodes are unavailable. Despite having many stale entries in the system, the DHT finds contact information for 93% of torrents by exploring approximately 50 DHT nodes. When compared with a tracker, DHT significantly suffers from the high response latency. However it can still succeed to provide contact information at the times when the trackers are unreachable. The experiments show that trackers in general provide more and faster information, but DHT increases the information availability of the whole system. In addition to that when used with a hybrid approach, DHT can help reducing the tracker load since peers can discover other peers via DHT and query the tracker more seldom.

A remarkable feature of DHT that should be highlighted is that, it offers the possibility to discover peers that are managed by different trackers but share the same content. The approach significantly increases the information availability when the overall BitTorrent system is considered by allowing content exchange among different swarms.

2.1.4.3. Peer Exchange Protocol

The peer exchange protocol (PEX) is a simple yet an efficient approach to improve information availability. Besides the centralized peer discovery method offered by trackers, most of the modern BitTorrent clients additionally provide decentralized peer discovery using Peer Exchange (PEX) in order to relieve the load on trackers and increase robustness against tracker failures. The PEX protocol leverages the knowledge of the peers that a peer is connected to; by asking them in turn for the addresses of peers they are connected to. The peers periodically share the contact information they have with each other. The information exchange directly takes place among the peers thus it does not cause any extra load on the tracker. PEX helps reducing the load of tracker and is a fast approach to expand local neighbourhood of peers.

In PEX protocol, peers gossip with each other in regular intervals with lists of *active* peers they know in order to uncover genuine peers interested in the same content. Unfortunately, this decentralized method of peer discovery first requires a peer to know at least one other peer using another method of peer discovery such as a centralized tracker or DHT. However once a peer obtains a list of peers in the swarm, PEX protocol does very well to discover other peers in the system. With its decentralized nature, PEX can help the swarm survive much longer in case of tracker failures, thus increasing the fault tolerance of the system. Furthermore in situations where the connectivity graph of the torrent starts getting disconnected due to any reason, PEX helps recovering the system back by merging the almost disjoint sub-graphs if peers from the disjoint sub-graphs exchange their contacts at least once.

It should be highlighted that being *actively* connected to a peer and knowing the address of a peer are different notions. In order to have reasonable amount of resource consumption, it is inevitable to have limitations on the number of the peers to connect; however a peer may keep the contact information of peers it met once in a buffer, despite not having an active connection with it, and share the buffered knowledge with other peers. Unfortunately a peer jeopardizes the efficiency of the peer exchange protocol by disseminating information about peers that are not in active connection with it; as it introduces the risk of sending addresses of peers that have already left the swarm, in other words sharing stale addresses. Even further a malicious peer may cause poisoning by injecting addresses of non-existing peers. Many other popular clients only exchange the contacts which they are actively connected to in order to ensure the distribution of peer addresses that are active in the torrent and to avoid stale peer addresses.

Currently, the PEX protocol does not have an official standard. Multiple versions of PEX protocol have been implemented and used by different clients, however peers should conform the same protocol to be able to exchange contacts. The known three versions of the protocol are: AZ_PEX which is implemented by the Azureus client, BC_PEX used by the BitComet client and UT_PEX which is preferred by many other clients such as libTorrent, µTorrent, Transmission, Mainline client and some others. According to the measurement study of [20], it is observed that 70% of peers support UT_PEX, 15-20% of peers support AZ_PEX and only about 5% of peers support BC_PEX. The peers that support these three PEX protocol is shown to account for approximately 95% of all the peers.

Although PEX is widely deployed in BitTorrent systems, comprehensive studies of PEX are not very common thus there is little knowledge about the behaviour of PEX in operational systems. According to the experiments performed in [20], PEX could improve the download performance of 40% of the tested torrents and the average reduction of the download time was measured to be around 7%. In addition to that, the authors of [20] point out the freshness of the PEX messages such that 30% of the PEX messages did not contain any stale addresses and the messages that contain at least 50% correct addresses is over 80% of all. This high ratio of freshness can be explained by the exchange of only active peers in the connection list. Despite having a high degree of redundancy, where some addresses can be delivered to a peer more than once, the message overhead of PEX is still at reasonable amounts.

2.1.4.3.1. UT_PEX Protocol

Among various versions of peer exchange protocol, UT_PEX is the one that is supported by most of the popular clients thus has been widely used in today's BitTorrent systems. The libTorrent client, which was selected as BitTorrent client for the experiments, also implements UT_PEX as peer exchange protocol.

The UT_PEX protocol has been implemented on top of the BitTorrent Extension Protocol [19]. During the extension protocol handshake, peers inform each other about the extensions they support thus clients supporting UT_PEX meet each other during the handshake. The rest of the protocol messages are sent with the extension message identifiers that are specified by the handshake. The protocol can be enabled or disabled according to peer's needs. If a certain number of peers are already discovered, the protocol can be disabled to avoid unnecessary overhead.

The peer exchange protocol has three important parameters to trade off between exchange efficiency and communication overhead. The parameters are as follows:

- *Time interval between peer exchanges*: The libTorrent client waits 2 minutes between two consecutive peer exchange messages. If the interval is too short the spread speed of the protocol will increase intuitively, however more bandwidth will be wasted with redundant messages. Although there is no official specification of PEX, most of the known clients have a waiting interval of at least one minute in order to avoid excessive increase in communication overhead.
- *Number of peers to exchange contact*: The number of peers that the client does peer exchange simultaneously is limited to 8 in default libTorrent implementation. It is also possible to have no limitation on the number of PEX candidates thus exchange contacts with everyone in the connection list. Although high number of candidates increases the efficiency of the protocol, the increase in the efficiency is marginal when the number of candidates goes over a reasonable limit but the increase in traffic is linear by the number of candidates.
- *Number of contacts sent in a PEX message*: The PEX messages sent by libTorrent client are limited to have 200 contacts at maximum whereas some other clients limit 50 contacts per message. The PEX message stores the contacts in binary format thus one contact requires 6 bytes of space which means that 200 contacts can easily be piggybacked to an already outgoing packet without causing much increase in the overhead.

In order to exchange peers, the client tries picking up 8 candidates that also support the UT_PEX protocol. The candidate selections are made uniformly random among the peers in connection list that are enabled for peer exchange. Unless the selected candidates violate the protocol specifications, they remain unchanged until they leave the system or disable peer exchange. If a candidate slot becomes available, a new peer is picked at random before the next peer exchange.

Table 1 : *UT_PEX protocol in libTorrent*

```
upon event ⟨ Init ⟩ do
```

```
        peer_exchange_interval = 2 mins
        max_pex_candidates = 8
        max_pex_message_size = 200
        last_saved_list = ∅
        ut_pex_initial = ∅
        ut_pex_delta  = ∅
end event

upon event 〈 Tick | every peer_exchange_interval 〉 do
        while selected_pex_candidates < max_pex_candidates && have_candidates_available do
                pick a new pex candidate (ut_pex enabled) at random from current connection
                list
                selected_pex_candidates++
        end while
        for all p ∈ selected_pex_candidates do
                mark p for sending pex message by the next outgoing message
        end for
        added = current_connection_list – last_saved_list
        removed = last_saved_list – current_connection_list
        ut_pex_delta = added & removed peers          // difference of connection list between two ticks
        ut_pex_initial = current_connection_list as added peers, no removed peers
        last_saved_list = current_connection_list
end event

upon event 〈 Sending any message to peer p〉 do
        if p is marked for sending pex message then
                if initial pex message to p then
                        send ut_pex_initial to p
                else
                        send ut_pex_delta to p
                end if
        end if
end event

upon event 〈 Receive pex message from peer p | added, removed〉 do
        insert peer addresses under "added" key to available peers buffer
end event
```

Table 1: UT_PEX protocol in libTorrent

At every 2 minutes the procedure that manages the peer exchange is executed. As soon as the candidates for exchange are selected, the PEX messages are prepared. The types of the PEX messages that can be sent to the candidates are:

- *Initial PEX message*: The initial PEX message is sent if it is the first message sent to a peer. Initial PEX message contains all the peers in the connection list. If connection list is larger than the allowed limit of the PEX message, the list is trimmed and the rest is sent later.
- *Delta PEX message*: Since the PEX candidates are kept constant once the connection lists are exchanged, then it can be kept up-to-date with delta PEX messages. The delta messages contain the changes in the connection list between two consecutive peer exchange ticks; the addresses of peers that are added to and removed from the connection list after the last exchange are transferred by the delta messages. The use of delta messages helps reducing the outgoing bandwidth.

The PEX messages are not sent immediately but delayed until another message is sent to the PEX candidate and the contacts to be exchanged are piggybacked to the already existing traffic to avoid causing extra load. However if the traffic between the peers is idle the client code is modified to force sending of the PEX messages. This modification will be discussed in detail in following sections.

Upon receiving a peer exchange message, the peer adds the contact addresses under the "*added*" key to the buffer of available peers. When necessary the discovered peers are picked randomly from the buffer and a connection is established. The list of "*removed*" peers is simply ignored by the libTorrent client as the client has its own mechanisms for removing connection with peers. In addition to that, removing connection depending on suggestions of other peers can make the client vulnerable to malicious attacks.

2.2. Kademlia

Kademlia is a communications protocol for peer-to-peer networks. It is one of many versions of a distributed hash table (DHT) but the one which is currently deployed and favoured by many BitTorrent clients.

The participating nodes in Kademlia form a structured overlay to store and lookup key-value pairs. The keys are 160-bit quantities usually selected as the SHA-1 hash of some larger data. The participants are identified with a quasi-unique node ID in the 160-bit key space generated by the node itself randomly. The key-value pairs are located on the nodes that have ids close to the key regarding the XOR metric of Kademlia. Each node maintains a routing table containing the contact information of a small number of other nodes in order to be able to perform lookups and route query messages appropriately.

In Kademlia, the distance between nodes or a node and a key is defined as the exclusive or (XOR) of their identifiers. The longer common prefix shared by two identifiers yields a smaller result of the xor operation thus meaning closeness. The important properties of xor metric are:

- the distance between a node and itself is zero
- it is symmetric, distances from X to Y and Y to X are the same
- it offers the triangle property: the distance from X to Z is less than or equal to the sum of the distance from X to Y and Y to Z

The properties above approve that xor is a valid and computationally cheap distance metric.

Every node in Kademlia maintains a routing table conventionally named as *k*-buckets. For each i, where i is $0 \leq i < 160$, a bucket is used to store contact information of nodes having distance between 2^i and 2^{i+1} from the node itself. *Figure 3* represents a node's routing table with 8-bit id space where each bucket stores nodes with corresponding distances. A node's contact information is formed by the IP address UDP port number pair and the node identifier. The buckets can store contact information of at most k nodes, where k is the system-wide replication parameter. It should be chosen carefully such that any given k nodes should be unlikely to fail at the same time interval in order to be able to perform lookups and avoid information loss in case of departures and failures as each object is stored at k-closest nodes to the object's identifier.

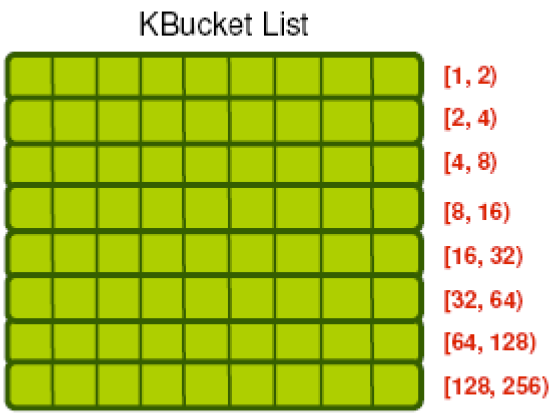


Figure 3: An example Kademlia node routing table with 8-bit id space (Figure from [14])

Considering the structure of k-buckets it can be noticed that the routing table gets more detailed for the identifiers closer to node's own identifier. Nodes know more about the closer nodes but have fewer contacts with far distances.

The routing table is maintained using the information retrieved only from the direct interactions between nodes, a node is considered alive only if a direct message is received from that node. Each k-bucket is sorted according to the timestamp of final interaction where the least recently seen node is located as the first element. When a node receives a message from another node, the appropriate k-bucket for the sender's identifier is updated as follows:

- If the sender is already in the bucket, the last seen timestamp of sender is updated hence moved to the tail of the bucket.
- If the sender is not in the bucket and if the bucket does not have k entries yet, the sender is inserted to the tail with fresh information.
- If the bucket is full (has k entries) the first element in the bucket which is the least recently seen is pinged. If a pong is received from old node the new node is simply discarded and old node is updated accordingly. If the old node fails to respond, it is removed from the bucket and the new node is inserted.

There is no restriction on the type of the message since they can all be considered as the evidence of aliveness of the sender. In situations where the routing table is not maintained spontaneously, a node can periodically perform random lookups in the range of inactive buckets or ping the nodes that are not active for a period of time.

The routing table of a node is maintained with the approach of removing the least-recently seen node. It is important to underline that a live node is never removed from the routing table. The preference for old contacts arises from the statistical analysis concluding that the longer a node has been alive the more probability it has to remain live [10]. The approach of keeping the oldest live contacts thus increases the chances of avoiding stale nodes in the routing table. Besides it helps avoiding denial-of-service attacks since a node's routing table cannot be polluted by malicious nodes in a short interval.

The Kademlia protocol consists of four remote procedure calls (RPC). They are:

- *PING*: Traditionally *ping* is used to determine whether a node is online if the node replies with a *pong* message.
- *STORE*: The *store* message is used for inserting the key-value pair to a node. The node receiving *store* message is expected to store the value for further retrievals.
- *FIND_NODE*: The message contains a 160-bit ID as an argument. The node receiving a *find_node* message returns the *k* nodes from its routing table closest to the given ID. The returned list may contain contacts from different buckets if the closest bucket does not contain *k* elements.
- *FIND_VALUE*: This message works similar to *find_node* messages and takes a 160-bit key as an argument. The recipient node returns the stored value for the key if it previously received a store message with the queried key. If the node does not store the value, it returns the *k*-closest contacts to the key.

All RPC packets are required to carry an RPC identifier assigned by the sender and echoed back in the reply. The identifier is a 160-bit random number and used to resist network address forgeries.

In order to store values in Kademlia, a node should be able to locate the *k*-closest nodes in the system to the given key. This procedure is named as *node lookup* and can be considered as the most important part of Kademlia. The lookup procedure is performed iteratively. A node starts by selecting α nodes closest to the target key, where α is the concurrency parameter with 3 as the optimal value [11]. The selected nodes are asked for their knowledge about *k*-closest nodes to the target key by sending asynchronous find node messages in parallel. The replies are used to update a temporary list storing the *k*-closest nodes to the target. In each iteration, the node queries α of the nodes from the temporary list which it has not queried before. The iteration stops when a round fails to discover a new node that is closer than the currently known nodes. To finalize the lookup the node queries each of the *k*-closest nodes that is has not contacted in earlier rounds. This process results with a set of *k* active contacts closest to the target identifier. During the iterations if a node fails to respond the query, it can simply be removed from the temporary list and the iterations may continue as normal. At each step the search comes at least one bit closer to the target. A node contacts only $O(\log(n))$ nodes for lookups in a system with *n* nodes. In value lookups, the iterations can terminate earlier if any of the queried nodes returns a value.

Figure 4 represents an example lookup initiation where node P performs a lookup for the identifier Q. The nodes A,B and C are the picked α nodes that are closest to Q and they are queried with find node messages asynchronously.

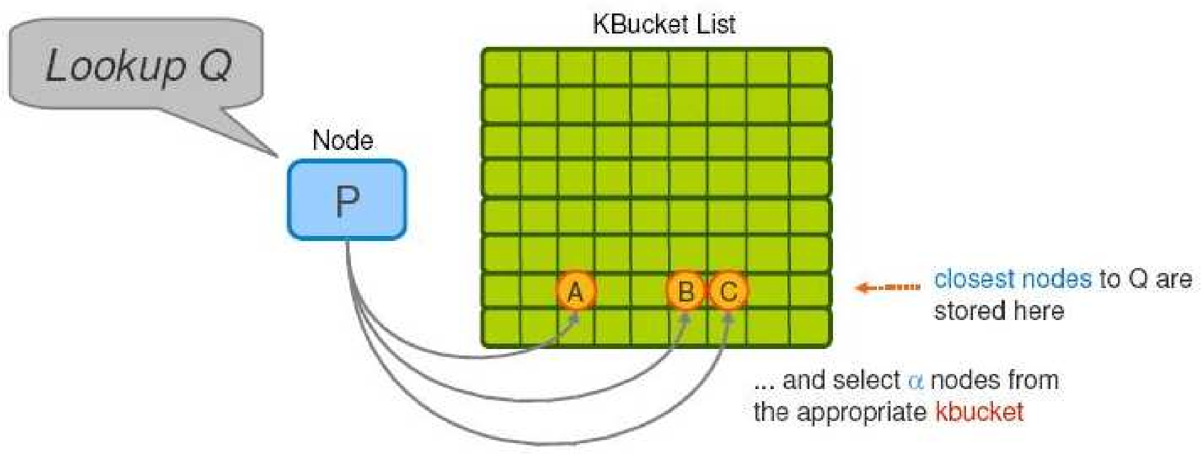

Lookup Q
Node
P
KBucket List
A
B
C
closest nodes to Q are stored here
... and select α nodes from the appropriate kbucket

Figure 4: A node lookup initial step (Figure from [14])

The following lookup iteration is illustrated in *Figure 5*. The information received from nodes A,B and C is inserted into the temporary list and the next iteration is performed with querying other nodes from the list. The received replies from A,B and C also serve the purpose of routing table maintenance by updating the last seen timestamps of the senders. The iteration stops when the temporary list remains same as the previous round and the lookup is finalized by querying the nodes that have not been queried previously.

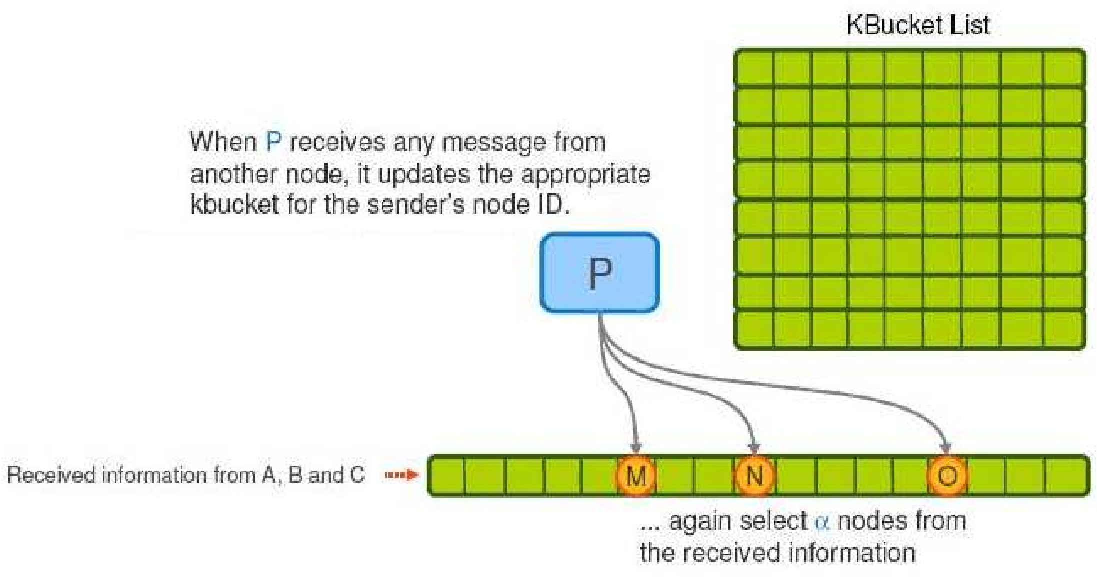

KBucket List
When P receives any message from another node, it updates the appropriate kbucket for the sender's node ID.
P
Received information from A, B and C
M
N
O
... again select α nodes from the received information

Figure 5: A node lookup iteration (Figure from [14])

To maintain the consistency of the system, nodes should replicate some of their information if they meet a new node that is closer to some of the keys they maintain. A lookup initiator should also store the key-value pair at the closest node seen if it did not return the value of the lookup.

Each node in the system is expected to re-publish the key-value pairs it has periodically in order to increase the persistency of information. A key-value pair in the system expires after 24 hours and the original publisher is expected to re-publish. The expiration rules limit the amount of stale information in the system.

A node can join the network through a known contact which is currently in the system. The node should insert the existing contact into its routing table and perform a lookup for its own identifier. The node should also perform random lookups within the range of each *k*-bucket. The random lookups will help populating the joining node's routing table and insert it to other nodes' tables as necessary.

2.3 Related Work

This section contains information about similar prior works to this thesis work that tries to improve performance of swarming systems. The common point of the considered works is the aim of increasing the overall system performance by improving the system and content availability in BitTorrent.

Neglia et al. [7] perform a large scale measurement study to investigate availability in BitTorrent. Their experiments show that BitTorrent swarms may seriously suffer from the tracker unavailability if trackers are left as a single point of failure of the system. The authors measure the effects of using multiple or replicated trackers and DHT-based distributed trackers. Their measurements show that both multiple trackers and distributed trackers help improving the information availability by increasing the chances of finding available trackers. The multiple trackers and DHT approach show complementary behaviour such that a combination of both provides high information availability with low information response latency. Use of multiple trackers helps balancing the load among trackers however it can significantly reduce the connectivity of the overlay formed by peers. The work in this thesis aims to increase the connectivity among different swarms by modifying peer behaviour.

Menasche et al. [15] consider the content unavailability as a fundamental problem of swarming systems. The content is considered to be available if either at least one seed is present or sufficiently many active leechers exist in the swarm to collectively make all constituent blocks of the content available. Considering the unpopular contents, the lately arriving peers may find the content unavailable since the availability of unpopular contents is mainly limited with presence of a seed or the publisher. According to their observations 40% of swarms have no seeds more than half of the time. In their work the authors consider BitTorrent as a queuing system and develop a model to quantify the content availability in swarming systems. Using the developed model the authors estimate the effects of bundling similar contents and validate their results through large-scale controlled experiments and analysis of real life torrents. The authors verify that distributing similar contents in a single package, for instance distributing entire season of a TV series, can significantly improve the content availability and increase the lifetime of unpopular content. The availability improvement of bundling may arise from different reasons such as changing user behaviour. However the model also suggests the same as bundling increases the busy periods, the length of uninterrupted intervals during which the content is available, of the system thereby reducing the dependency on the initial publisher. The authors also find out that in contents with a highly unavailable publisher; bundling can reduce the download time of the unpopular content even though more content is downloaded. The download time improvement can arise when the waiting time, the time spent without downloading content due to unavailability, highly dominates the service time as the improved availability by means of bundling can amortise the download time of the extra content. It should be noted that two different forms of bundling exist which are pure bundling and mixed bundling. Pure bundling forces the consumer to have the entire bundle or none whereas mixed bundling gives the chances of selecting parts of the package. Both forms of bundling can improve content availability and increase the lifetime of torrents however mixed bundling is more common in recent distributions.

Peterson et al. [16] introduce a new content distribution system that differs from BitTorrent with its capability of managing swarm. Antfarm tries to improve the performance of content distribution by viewing it as a global optimization problem, where a coordinator directs bandwidth allocation at each peer considering the bandwidth constraints and swarm dynamics in order to reduce the download latencies of participants. The main idea of Antfarm becomes clearer when peers contribute in different contents' distribution so that the coordinator can distribute peer bandwidth among multiple swarms effectively to avoid content unavailability in each swarm. The BitTorrent protocol, having an unmanaged swarming architecture, may perform well enough within a single torrent. However the local bandwidth allocation algorithms of BitTorrent may lead to starvation in multi-torrent settings as it does not consider the varying swarm dynamics. In order to make it possible to evidently determine the swarm dynamics by the coordinator, Antfarm protocol makes use of unforgeable tokens that forces the participants to divulge their upload contributions. While the coordinator allocates bandwidth among competing swarms; the peers are still allowed to use local optimizations similar to BitTorrent protocol such as tit-for-tat policy, optimistic unchoking or rarest-first pieces selection policy. The evaluation results show that Antfarm and BitTorrent perform similar when a single swarm is considered. However when distributing multiple contents, Antfarm successfully distributes available bandwidth among swarms whereas unpopular swarms may starve using BitTorrent protocol. One drawback of the Antfarm is the scalability of the protocol where the coordinator can become a bottleneck as it requires more resources than a standard BitTorrent tracker in order to determine swarm dynamics and direct bandwidth allocation among swarms. The protocol is designed to enable hierarchically distributed coordinators in order to alleviate the bandwidth demands placed on coordinator.

Dàn et al. [17] in their work, point out the efficiency degradation in small swarms with few participating peers due to the low availability. According to their measurement, thousands of swarms are detected whose performance can significantly be improved using distributed swarm management algorithms. The authors introduce two algorithms that aim to improve performance of multiple small swarms of the same content by increasing the connectivity of disjoint swarms. The first algorithm presented is dynamic swarm management (DSM) which works at the tracker level. DSM achieves performance improvement by identifying and merging small swarms of the same content, however it also supports splitting considerably large swarms in order to ensure load sharing among trackers. In order to enable swarm management, each tracker periodically picks another tracker in order to perform pairwise load balancing. In load balancing algorithm trackers exchange information of the contents that they both manage and determine which tracker should be responsible from which peers. The second algorithm discussed in the paper is randomized peer migration (RPM) where a small fraction of peers migrate between different swarms and disseminate contact information of peers from different swarms using PEX protocol. RPM only requires modification of peer behaviour and the communication overhead is increased only by the register/unregister messages used for migrating between trackers. The migrating peers increase connectivity among multiple swarms, virtually increasing the size of small swarms thus improving the overall throughput. Both algorithms are shown to help improving the performance of small swarms with only a modest increase in communication overhead. Random peer migration will be observed and evaluated in more detail in the following sections.

3. DISTRIBUTED ALGORITHMS DESIGN

This section contains design details of the dynamic swarm management algorithms. It first discusses implemented lightweight algorithms for increasing information availability, Random Peer Migration (RPM) and Random Multi Tracking (RMT). Additional algorithms such as Baseline and Pick Biggest Swarm (PBS) are also discussed as a criterion for evaluation purposes. Later the improved PEX protocol whose main aim is to facilitate better mixing among swarms is mentioned.

Table 2 contains the notations that are frequently used for modelling BitTorrent systems in following sections.

$x(t)$	number of peers in torrent at time t
$l(t)$	number of leechers in torrent at time t
$s(t)$	number of seeds in torrent at time t
λ	the arrival rate of peers
μ	the uploading bandwidth of peers (*assuming all peers have same upload bandwidth*)
d	the downloading bandwidth of peers (*assuming all peers have same download bandwidth and* $d \geq \mu$)
T	average time required for download completion
Δ	average time that seeds stay in swarm after download completion
R	set of trackers that track the torrent
n	number of trackers that track the torrent $(\lvert R \rvert)$
β	willingness parameter for performing swarm management
p	maximum number of peers that a peer can connect to (*size of the connection list*)
c	maximum number of peer exchange candidates

Table 2: The frequently used notation for modelling BitTorrent systems

3.1. Swarm Management Algorithms

In order to relieve the load on a single tracker and increase resilience against tracker failures, BitTorrent protocol allows a single torrent to be tracked by multiple trackers. To have the load distributed fairly and avoid excessive overhead on trackers, a peer is allowed to contact with only one tracker per content. Therefore the peers interested in same torrent may form disjoint swarms resulting with a split overlay in which peers in different swarms are not aware of each other. According to previous measurements increasing the size of a swarm is known to increase performance of the BitTorrent protocol if other parameters are left constant. Consequently the clustering due to use of multiple trackers causes the protocol to perform worse than the peers were in a single swarm. Despite causing performance degradation, use of multiple trackers is inevitable since it increases the system availability against tracker failures and relieves the heavy load on trackers.

The swarm management algorithms aim to increase the protocol efficiency while preserving the resilience and load balancing properties offered by use of multiple trackers. Basically the algorithms introduce peers from different swarms to each other through peer exchange protocol thus increasing swarm sizes virtually and performance correlatively.

Random Peer Migration (RPM) and Random Multi Tracking (RMT) require modification only to peer behaviour. In order to achieve mixing of swarms, a fraction of peers interact with more than one swarm and distribute contact information among swarms using the PEX protocol. The algorithms do not require trackers to maintain extra state information and simply take place between the existing traffic between peers. The only increase in communication overhead is due to interaction between peers running the algorithms and trackers such as register, unregister and scrape messages. The number of peers that perform the mixing is independent of the swarm size and can be controlled by the protocol parameter therefore the mixing can be achieved with only a modest increase in communication overhead.

To gain better understanding of swarm management algorithms two additional peer behaviours should also be mentioned. First is the baseline, namely vanilla, peer behaviour which is the default behaviour of BitTorrent clients and second is Picking Biggest Swarm (PBS) whose behaviour is obvious from the name. These two algorithms can be considered as opposite ends that represent pure load balancing and maximum performance respectively. The mixing algorithms can be placed between the two where they aim to increase performance as much as possible while preserving the load balancing and resilience against tracker failures.

3.1.1. Baseline

The vanilla peer represents the default behaviour specified in BitTorrent protocol. Given a set of trackers, one tracker is picked uniformly random and the peer sticks to that tracker until departure unless the tracker fails.

With vanilla behaviour the swarms are expected to be equally sized on long term average thus achieving fair load distribution. However as mentioned earlier the download performance of the torrent is at minimum, especially for small torrents, due to partitioning of the overlay. The communication overhead of baseline peer is at minimum when compared to other algorithms as each peer only associates with a single tracker.

The swarm management algorithms are developed based on the vanilla behaviour however depending on the defined protocol parameters peers may exhibit different behaviour to mix the swarms.

3.1.2. Picking Biggest Swarm

With PBS protocol, a peer scrapes all trackers in given list before associating with one of them. The scrape queries are similar to tracker announces however scrapes are replied only with state information of the swarm, number of leechers and seeds in the swarm instead of a subset of peers. Scraping all trackers, the peer can decide to join to the biggest swarm so that it can maximize the download performance. In larger swarms a peer has higher probability of receiving pieces as piece diversity and peer contribution is higher when compared with smaller swarms.

The size of a swarm is simply defined as the number of peers in it, sum of seeds and leechers. Alternative approaches such as prioritizing the number of seeds can also be considered as available seed capacity is an important factor on download performance. Currently the available seed capacity is considered only as tie-breaking condition for equally sized swarms. If swarms are still equal, a tracker can be picked with a pre-determined approach such as picking the first tracker in the list. This selection is in general performed by the publisher as it picks one from the set of empty swarms.

If all peers exhibit the PBS behaviour, it is obvious that all peers will be grouped in a single swarm which is selected by the publisher at start. Grouping all the peers in a single swarm, the pick biggest swarm algorithm achieves to maximize the download performance. On the contrary the load balancing and resilience properties are sacrificed.

Upon failure of the biggest swarm, all peers are expected to migrate to another swarm which is empty at that time. The first peer that detects the failure picks another swarm with the deterministic approach.

The deterministic approach is favoured to avoid scenarios such as peers joining to different swarms at the same time even though it has a low probability. As soon as a single peer joins a swarm, which is enough to make it the biggest, remaining peers will join the same swarm with PBS algorithm. This approach is similar to back-up trackers.

When considered for a single content the load balancing property is totally diminished as all overhead is loaded on a single tracker. However looking at the bigger picture with multiple contents, the load will be fairly distributed among trackers since contents will be associated with different trackers.

When compared with the vanilla behaviour, the only increase in communication overhead is the tracker scrapes. The load increase on each tracker is one scrape per peer which is reasonable.

It is obvious that using pick biggest algorithm, the BitTorrent performance can be maximized. However use of PBS does not ensure system availability and load balancing among trackers.

3.1.3. Random Peer Migration

The random peer migration protocol achieves mixing among swarms by means of the peers that migrate between swarms. According to the protocol parameters some share of the peers migrate between swarms at random. Upon migration, the arriving peer distributes the contact information of peers in the previous swarm to the peers in the newly arrived swarm over the peer exchange protocol. The peer migration between swarms is a simple, cheap yet an effective way of swarm mixing.

The RPM protocol is an extension of the vanilla behaviour. Different from vanilla, peers decide whether to migrate or not periodically. Upon downloading or uploading $1/(\beta(|R|-1))$ portion of the whole content the peer performs probability check for migration where β represents the migration willingness of peers. For instance considering a torrent with 2 swarms and β value set to 2, peers will perform migration check upon transferring, sum of total upload and download, half of the content. If upload and download rates of peers are assumed to be equal, the probability check will be performed approximately 3-4 times before completing download. This approach removes the dependency of migration frequency from content size. Increasing β decreases the portion size thus increases the frequency of probability checks for migration, causing peers to migrate more often.

Upon transferring data with size of the determined portion, a peer decides to migrate to another swarm with probability $1/x_r$ where *r* represents the current swarm thus x_r is the number of peers in current swarm. The peers migrate at a similar rate for wide range of swarm sizes by the protocol design thus making the protocol overhead independent of the swarm size.

If the peer decides to migrate, another tracker r' is picked uniformly random from the set of remaining trackers, or in other words from the set of all trackers except r. The tracker r' is scraped to avoid migrating to empty swarms. If r' is empty the migration is cancelled otherwise the peer migrates to swarm r'. The migration process is simple such as unregistering from tracker *r* and registering to r'.

While migrating from swarm r to r', the peer's connection list is filled with addresses of peers in swarm r. Upon registering to tracker r', a set of peers in swarm r' is also obtained. Knowing peers from different swarm, the migrating peer can mix swarms through peer exchanges. With a slight modification on existing PEX protocol, the migrating peer may either decide to introduce peers from previous swarm to the peers in the new swarm or vice versa. It is also possible to leave the direction of exchanges random. The improved PEX protocol that provides control on the direction of the information flow will be discussed in Section 3.2.

The connection list size or the number of peers that a migrating peer carries represented by p, is another important property that affects the efficiency of mixing. As it is obvious transferring more peers per migration or increasing p, increases the mixing efficiency of the protocol.

- *Protocol Overhead*: The only overhead increase caused by RPM is when peers decide to migrate. For each migration, a peer performs a tracker scrape and two announces for unregistering and registering. The increase in overhead is positively correlated with the migration frequency therefore directly proportional to $\beta(|R|-1)$. As the load increases linearly with the willingness parameter it should be determined suitably to avoid excessive overhead while achieving good mixing.
  Assuming that the performances of different swarms are equivalent, peers will perform migration checks with same frequency as they will transfer the defined amount of data at similar durations. Having the same willingness parameter, the outgoing number of peers from each swarm will be the same on long time average as migration checks are performed at similar rate. It should be re-stated that the probability check for migration makes the protocol independent from the swarm sizes. As the selection of swarm to migrate is uniformly random, the outgoing peers will be distributed fairly to the swarms. Since each swarm has the same number of outgoing peers distributed equally on others, the number of outgoing and incoming peers of each swarm will be equivalent. Consequently one can observe that RPM protocol does not cause a load imbalance on existing swarms but simply achieves mixing.

Even though mixing the swarms through peer migrations seems a trivial solution, the protocol rules should be defined suitably to be able to apply on various torrents with different parameters such as swarm size or file size and under changing swarm dynamics.

The RPM algorithm is compatible with existing BitTorrent protocol and it can simply be disabled by setting willingness parameter to 0 if the client associates with private trackers.

3.1.4. Random Multi Tracking

The random multi tracking protocol achieves mixing among swarms by associating some peers with more than one tracker upon arrival. According to the protocol parameters some share of the peers join several swarms at random upon arrival and perform mixing among swarms through peer exchanges by sending contact information of peers in a swarm to the peers in other swarms.

Alternatively if all peers associate with all trackers the protocol performance can be maximized while preserving the resilience property. However the mentioned aggressive approach causes the overall tracker load to increase proportional to $x * (|R|-1)$. Therefore the number of multi-tracking peers should be suitably determined such that the mixing between swarms is achieved at a low overhead.

The RMT protocol works as follows. Upon joining a torrent, the peer scrapes all available trackers to determine the number of peers in the torrent, represented with variable x. Considering the special case of publisher, where $x = 0$, the peer can either join one of the trackers at random or register to all. If the torrent is not empty so that $x > 0$, the peer can decide to associate with k randomly picked trackers from available n trackers with a probability of $min(1, \frac{n\beta}{kx})$. Otherwise a single tracker is selected at random which is the same as vanilla behaviour. The protocol parameter β is now used to represent the willingness of peers to multi-track, similar to migration willingness of RPM protocol. The probability check generates close number of multi-tracking peers for wide range of swarm sizes thus making the protocol overhead independent from the swarm size.

Upon deciding for multi-tracking, the number of peers requested from trackers is divided by k so that equal number of peers is obtained from each swarm so a multi-tracking peer connects to p/k peers from each swarm on the average where p is the size of connection list. Knowing peers from different swarms, the multi-tracking peer can mix swarms through peer exchanges by introducing peers of different swarms to each other. The connection list size, p, obviously affects the efficiency of mixing as exchanged addresses increase proportionally by p.

- *Protocol Overhead*: The overhead increase caused by RMT is due to the scrapes performed by each peer on start-up and the overhead caused by multi-tracking peers. When the system is in steady state, the average number of multi-tracking peers in the system is expected to be $n\beta/k$. Each multi-tracking peer associates with k trackers therefore the load caused by multi-tracking peers is directly proportional to $n\beta$. The protocol overhead can be adjusted by the willingness parameter however it should be determined suitably to avoid excessive load increase as the overhead increases linearly by willingness. It should be noted that the increase in overhead is fairly distributed among trackers as multi-tracking peers pick k trackers at uniformly random.

Even though mixing the swarms through associating peers with multiple swarms seems a trivial solution, the protocol rules should be defined suitably so that excessive increase in communication overhead can be avoided while good mixing is still achieved. In addition to that the protocol can be applied to wide range of torrents with different parameters such as torrent size or number of swarms.

The RMT algorithm is also compatible with current BitTorrent protocol as vanilla behaviour can be obtained by setting willingness to 0 when necessary. In addition to that a peer can be forced to associate with k trackers through parameter settings for experimental use.

3.1.4.1. Deciding a Suitable k value for RMT

As the overhead is independent of k, the protocol can be evaluated to determine the optimal value of k that provides the maximum mixing while keeping the protocol overhead constant. Considering a torrent with n swarms, the mixing contribution of multi-tracking peers can be calculated as follows. Assuming that a peer is allowed to connect up to p peers, it is known that a multi-tracking peer maintains information of p/k peers per each swarm. Let c represent the maximum number of peer exchange candidates per interval then a multi-tracking peer exchanges contacts with c/k peers from each swarm as the candidates are picked at random from the set of available peers. Then the amount of external contact information injected to a single swarm by a single multi-tracking peer can be calculated as:

$$\frac{c}{k} * \frac{(k-1)p}{k} \tag{1}$$

as the contacts known from remaining $(k-1)$ swarms are sent to c/k peers in the particular swarm. Since a multi-tracking peer connects to k swarms, the total amount of inter-swarm information transferred by a single multi-tracking peer would be:

$$\frac{c}{k} * \frac{(k-1)p}{k} * k \tag{2}$$

As it is obvious changing k also changes the overhead caused by a single multi-tracking peer, however the protocol keeps the overhead constant by varying the number of peers that apply multi-tracking. If the overhead of a vanilla peer associating with a single tracker is assumed as 1 unit of load, a multi-tracking peer that connects to n trackers generates n units of tracker load and consequently connecting to k trackers causes k units of load. Therefore for every k=n multi-tracking peer, one can have n/k peers that connect to k trackers. For instance in a torrent with 3 swarms, mixing the swarms using 3 peers that associate with 2 trackers causes the same overhead with 2 peers associating with 3 of the trackers. Therefore the total amount of inter-swarm information transfer under the same load can be calculated as:

$$\frac{c}{k} * \frac{(k-1)p}{k} * k * \frac{n}{k} \tag{3}$$

Simplifying the equation (3) the protocol performance can be evaluated as below:

$$c * p * n * \frac{(k-1)}{k^2} \tag{4}$$

The equation (4) shows that the protocol performance increases as k value is lowered therefore selecting k as 2 is optimal for mixing.

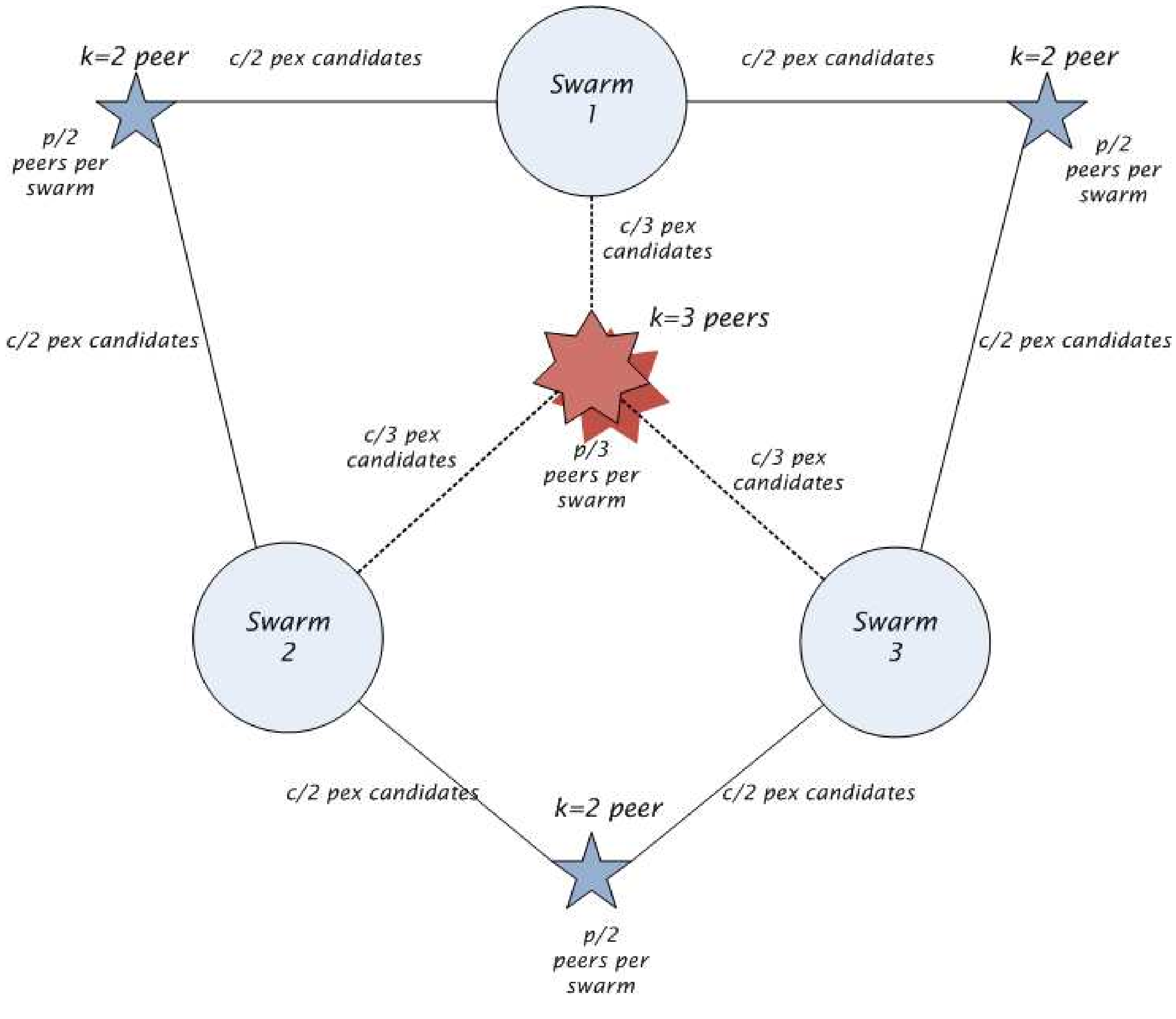


Figure 6: Multi-tracking peers in a torrent with 3 swarms

Figure 6 illustrates the developed model on a torrent with 3 swarms. Each edge between peers and swarms represents an inter-swarm information exchange. For k=2 peers each edge carries $\frac{c}{2} * \frac{p}{2}$ contact addresses whereas for k=3 the amount is $\frac{c}{3} * \frac{2p}{3}$ as information from 2 swarms are injected to a single swarm. It can be noticed that when k=2 the number of multi-tracking peers is 3 however for k=3 only 2 peers can be used to preserve the overall load.

The model discussed above is based on two assumptions:

i. The connection list size is limited so that increasing k decreases the number of peers known per swarm.

ii. The maximum number of candidates for peer exchange is limited so that increasing k decreases the spread speed of peer exchange per swarm.

The assumptions above can be relaxed as follows:

i. If p is much greater than torrent size $p \gg torrent\ size$, changing k would not affect the number of contact addresses known per swarm. Therefore equal number of peers can be known from each swarm independent of k.

ii. If c is much greater than torrent size $c \gg torrent\ size$, changing k would not affect the number of peer exchange candidates selected per swarm. However the maximum number of PEX candidates is still bounded by p as peer exchange is only performed between actively connected peers. On the other hand considering the exponential dissemination of PEX messages, changing k may not dramatically harm the spread speed of the inter-swarm information for small torrents. Algebraically, the spread speed of the newly injected contact information can be approximated as $\frac{c}{k} * c^i$ where *i* represents the number of PEX iterations, if redundancy in peer exchanges are ignored optimistically. According to equation for small torrents the k parameter can be avoided as all peers in swarm will receive the new contacts after a small number of iterations. Therefore depending on the size of the torrent, the effect of k on the speed of information spreading can be relieved.

Relaxation of the assumptions above, the result implied by the model can change. Considering assumption i, the number of peers known per swarm should be changed to p instead of p/k. If assumption ii is relaxed, the amount of inter-swarm information injected to each swarm should be calculated with factor *c* instead of c/k. Relaxing one of the assumptions above the equation becomes:

$$c * p * n * \frac{(k-1)}{k} \tag{5}$$

On contrary with equation (4), equation (5) increases with *k* therefore setting *k* to *n* maximum mixing can be achieved.

The information derived from the model can be summarized as follows. For small torrents or high *p, k* should be maximized hence should be set to *n*. On the contrary for large torrents or low *p*, setting *k* to 2 might be the right choice.

3.2. Improved PEX Protocol

The swarm management protocols such as Random Peer Migration (RPM) and Random Multi Tracking (RMT) leverage the peer exchange protocol to distribute peers' contact information among different swarms. However the plain peer exchange protocol fails to fully utilize the mixing efficiency when multiple swarms exist for a single torrent, since the protocol does not take swarm memberships into account while exchanging peers. In order to increase connectivity among multiple swarms when combined with the algorithms, an improved peer exchange protocol that considers swarm membership during peer exchange has been developed. The improved PEX protocol is given the name LT_PEX where the extension name is prefixed by abbreviation of libTorrent client to prevent collisions in extension names as mentioned in protocol specification.

When multiple swarms exist, a peer discovers peers from own swarm via tracker queries. When algorithms that pioneer interaction among disjoint swarms are used, the peers find the opportunity to meet peers from other swarms, namely *external peers*. In order to make it possible for algorithms to distribute information between different swarms, the peer exchange protocol should be capable of differentiating peers according to their swarms and controlling the direction of information flow among swarms. In addition to that it is even better for ordinary peers to prioritize discovery of external peers during peer exchange since peers in same swarm can already be discovered via tracker queries meanwhile. The improved PEX protocol, taking advantage of swarm membership awareness, provides a framework to swarm management algorithms to have control over the peer exchange and effectively mix the peer information between disjoint swarms.

In order to gain awareness of swarm membership, the extension protocol handshake has been slightly modified such that the peers notify each other about the swarms they belong to during the handshake. In current implementation of LT_PEX, the announce url's of the trackers are used to uniquely identify the swarms. A new item, with key named *swarmid*, has been added to the extension handshake message that transfers the swarm information. An example extension protocol handshake message carrying swarm information between peers can be observed in *Figure 7*.

Extension Handshake		
m	**Dictionary**	
	UT_PEX	1
	LT_PEX	2
p	50529	
v	"libTorrent-0.12.6"	
swarmid	"http://10.254.1.1:6969/announce"	

Figure 7: An example of extension protocol handshake message carrying swarm information

Considering the handshake message above, the peer sending the message supports two extensions which are UT_PEX and LT_PEX. The key p is used to represent the local TCP listen port of the peer. The key v is an alternative way to transfer client name and version. The swarm identifier is transferred under the swarmid key. It should be noted that the additional keys such as p, v and swarmid are optional and may not be supported hence discarded by some clients.

The swarm membership discovery is not only limited with the extension protocol handshake since some clients may not support the new protocol. The developed client leverages two additional ways to discover the swarm that peers belong to:

- *Tracker queries*: The tracker queries can be used as an alternative source for gathering swarm membership information without showing any extra effort. The peer set sent in a tracker reply is simply marked with the tracker's announce url as the swarm of the peers is apparent.
- *Improved peer exchange messages*: As another source of peer discovery, the peer exchange can additionally transfer swarm membership information. The clients supporting LT_PEX protocol classify the peers according to their swarms during exchange. Thus the receiving end of the peer exchange gets the peer set tagged with their swarm information.

An additional property of methods mentioned above is the swarm information of the peer is learned without initiating a connection thus facilitating the client to develop strategies while selecting a peer to connect from the list of available peers.

In cases where the peer's swarm cannot be learned, peer's swarm field is marked as *unknown* temporarily. If the swarm information of the peer is collected with the additional ways, the *unknown* field is updated. However if the swarm information of a peer cannot be gathered by any means, the unknown peer is behaved as a member of swarm named *unknown*. The *unknown* swarm is neither different from nor the same with any given swarm. For instance, if the peers from own swarm are requested the *unknown* peers are behaved the same with the external peers. If external peers are requested, the unknown peers are considered as members of own swarm.

Table 3 : *LT_PEX protocol*

```
upon event 〈 Receive extension protocol handshake msg from peer p | swarm_id 〉 do
        p.swarm_id = msg.swarm_id
end event

upon event 〈 Init 〉 do
        peer_exchange_interval = 2 mins
        max_pex_candidates = 8
        max_pex_message_size = 200
        peer_exchange_direction = either prefer_own_swarm or avoid_own_swarm or random
        for all p ∈ current_connection_list do
                p.peer_transfer_history = {p}
        end for
end event

upon event 〈 Tick | every peer_exchange_interval 〉 do
        if prefer_own_swarm then
                try picking all pex candidates (8) (lt_pex enabled) at random from current
                connection list but prioritizing the peers from own swarm
        else if avoid_own_swarm then
                try picking all pex candidates (8) (lt_pex enabled) at random from current
                connection list but prioritizing the external peers (peers from other swarms)
        end if
        for all p ∈ selected_pex_candidates do
                possible_peers_to_send = current_connection_list – p.peer_transfer_history
                pick peers from possible_peers_to_send but first picking peers external to p
                p.peer_transfer_history = p.peer_transfer_history ∪ picked_peers
                bencode picked_peers as p.lt_pex_message
                mark p for sending pex message by the next outgoing message
        end for
end event
```

```
upon event ⟨ Sending any message to peer p ⟩ do
        if p is marked for sending pex message then
                send p.lt_pex_message to peer p
        end if
end event

upon event ⟨ Receive pex message from peer p | added, removed ⟩ do
        p.peer_transfer_history = p.peer_transfer_history ∪ added
        insert added peers to available peer buffer  //add peers from buffer at random if peers needed
end event
```

Table 3: LT_PEX protocol

Different from the UT_PEX protocol, a new parameter called *peer_exchange_direction* has been added. While selecting candidates for peer exchange, a peer may prefer peers from own swarm or external peers that can be set by parameter values *prefer_own_swarm* and *avoid_own_swarm* respectively. If no value is assigned for exchange direction the candidate selection is pure random.

In the beginning of every peer exchange interval, the PEX candidates are picked randomly from the set of peers if specified by the peer_exchange_direction parameter. If the preferred set does not have enough members to fulfil available slots for peer exchange, remaining candidates are picked from the whole connection list. Since PEX candidates are replaced at every interval the spread speed and efficiency of the protocol is expected to be higher than UT_PEX protocol. The protocol limits the number of outgoing PEX messages but not the incoming messages. Fortunately the number of incoming messages per interval is equal to the number of outgoing messages on the average since the candidate selections are uniformly random. Possible cautions such as disconnecting peers that cause excessive peer exchange traffic can be considered to avoid malicious peers.

In order to avoid sending redundant addresses to the candidates, a transfer history is kept for each peer. The addresses that are sent to or received from a peer are inserted to the transfer history of that peer. Before sending an exchange message to a particular peer, the transfer history is subtracted from the current connection list. Firstly the peers that are external to the receiving peer are picked randomly from the remaining set and inserted into message. If available space is left, contacts from the same swarm are appended. In other words the selection policy for which peers to send in a PEX message is prioritizing the peers that are in different swarms with the remote end. This selection policy increases the connectivity among different swarms. As mentioned above, the PEX messages are also used for transferring swarm membership information therefore the peers are sent in groups by their swarm identifiers. It should be highlighted that PEX messages sent to each peer are generated separately by considering swarm membership of each candidate. Intuitively sending different contacts to each candidate increases the entropy of the system and helping faster dissemination of contacts.

As mentioned earlier if the size of the connection list goes above a certain limit peer exchange protocol is disabled to prevent unnecessary traffic. However different from UT_PEX, LT_PEX client only stops receiving messages but continues disseminating peer addresses even after the protocol is disabled. The idea behind this modification is helping other peers altruistically even though the peer itself does not need to discover any other peers. Apparently this modification increases the efficiency of the protocol but also causes extra communication overhead.

When a peer receives a peer exchange message, the received peers under *added* key are inserted to the buffer of available peers. Also received peers are added to the transfer history of the peer to avoid redundancy. Different from UT_PEX messages, an LT_PEX message contains swarm identifier and peer set pairs under the added key. The peer sets are encoded as strings in compact form. Optionally the peers can append their own swarm information to the PEX message with *ucs (update current swarm)* key in case any updates are necessary.

Improved PEX Message		
added	**Dictionary**	
	“http://10.254.1.2:7070/announce”	10.0.0.1:50001
		10.0.0.2:50002
		10.0.0.3:50003
	“http://10.254.1.1:6969/announce”	10.0.1.10:50260
		10.0.1.11:50261
		10.0.1.14:50264
ucs	“http://10.254.1.1:6969/announce”	

Figure 8: An example of improved PEX message

An example of improved peer exchange message can be examined in *Figure 8*. The *ucs* key carries the swarm identifier of the message sending peer. The sent peers are grouped by their swarm identifiers and peers that are external to the receiving side are prioritized. The set of peers are sent as a compact string where each address and port pair is represented by 6 bytes.

In order to efficiently use in coordination with algorithms like Random Peer Migration and Random Multi Tracking, the peer exchange direction is set to pick candidates from own swarm. The policy for selecting peers to include in the PEX message can have negligible effect with the current parameters since the size of a PEX message is much higher than the size of connection list. However the policies for candidate selection are still important by facilitating better mixing of disjoint swarms.

4. FRAMEWORK IMPLEMENTATION

Evaluating BitTorrent performance is hard because it is not easy to perform repeatable large-scale experiments that can be a representative of real BitTorrent systems. On the other hand, BitTorrent simulations may show poor accuracy when compared to real systems as simulation assumptions may fail to capture all the protocol details.

This section discusses the design of the framework where the controlled content distribution experiments with BitTorrent are performed on multiple computers. It contains key modifications on a BitTorrent client and notable techniques that made it possible to run experiments that are similar to real life. This section can be considered as the road map to be followed in order to perform content distribution experiments with a BitTorrent client. Finally the implementation details of testbed components are given.

4.1. Selecting BitTorrent Client

In order to make it possible to run many instances of the BitTorrent client on each computer involved in the experiments, a light-weight client has to be chosen. libTorrent [18], developed by Jari Sundell, is a BitTorrent library written in C++ for *nix focusing on high performance and good code. The library increases performance by transferring data directly from file pages to the network stack that puts the client one step ahead of other implementations. The rTorrent client developed on libTorrent runs on the terminal screen by making use of the ncurses library. When combined with the screen application in Linux, multiple instances of rTorrent can easily be run and controlled on a single computer while keeping the resource consumption at a reasonable level. Running rTorrent client in screen application also makes it suitable to prepare scripts that remotely controls the client using the SSH tool. The generic design of rTorrent enables users to control variety of parameters without the need for code modification. For instance a user can log down torrent statistics (e.g. download time, average download rate) or modify client behaviour using predefined events by adjusting the option file of the client. In addition to that, readable source code and loosely coupled design of libTorrent help the developers to modify the client behaviour according to their needs with modest amount of effort.

Despite being an appropriate client to be used in content distribution experiments, libTorrent and rTorrent contain some defects and features that have to be removed to increase the validity of the experiments. The necessary modifications to use rTorrent in experiments are as follows:

- The piece selection policy of libTorrent favours to download pieces of same rarity sequentially and randomizes the selection process at 16 pieces on the average. The rationale for this behaviour is to decrease the cost of I/O access with accessing the disk sequentially. However it significantly reduces the overall performance of the system if all participating clients exhibit the same behaviour; causing the piece diversity of the swarm to be less than expected. Considering a distribution scenario with a single seed, the sequential piece selection policy causes all participants to request the same piece after some time which fails to utilize upload bandwidth of peers due to reduced piece diversity and limits the download performance by the upload bandwidth of the seed. To avoid this performance degradation the selection process is set to be randomized at every new piece, consequently increasing the entropy of pieces in the swarm.
- In default implementation of libTorrent some of the protocol messages, such as *HAVE* and *PEX* messages, are piggybacked to the data transfer messages in order to reduce the communication overhead. *HAVE* and *PEX* messages never force the client to send a new packet but instead they are delayed until a data transfer or heartbeat message is sent. When a

pair of peers that do not have active transfer with each other is considered, *HAVE* messages will be delayed until the next heartbeat message which will cause the peers to be unaware of the newly available pieces possessed by its neighbour for approximately 2 minutes. The effect of this blackout period among peers is negligible when the content download time is around an hour; however for an experiment in which the download time is around 10-20 minutes, this effect would certainly harm the system performance since the peers that are not interested in each other will not be able to share the new pieces they download with each other for around 2 minutes, which is a considerable amount of time when compared to the total download time. In order to overcome this problem, *HAVE* and *PEX* messages are forced to be sent if the connection between two peers is idle. Otherwise, when there is an active transfer between peers, these messages are anyhow piggybacked to the piece messages. This modification prevents *HAVE* messages from being delayed, thus the piece availability of the system increases as all the peers are updated with fresh information about the pieces available in their neighbourhood. The modification yields an improvement of the system throughput. The instantaneous transfer of *PEX* messages increases the spread speed and efficiency of the protocol intuitively. Even though this modification causes an increase in the communication overhead of the whole system, it is necessary as it makes the effects of swarm size and PEX protocol more visible for the downscaled experiments.

- The Extension Protocol [19] in rTorrent contains a defect such that a peer can be considered as supporting an extension although it does not actually. In Extension Protocol, each extension is assigned an extension number where a value of 0 represents that the protocol is disabled/not supported by the client. However libTorrent code fails to handle the case where the number of an extension is set to 0, assuming the client to support the extension without checking the assigned number. The client has been modified to carefully handle the extension numbers where the extensions having an extension number set to 0 are considered as not supporting or currently disabled the extension. The defect was causing performance degradation in peer exchange protocol (PEX), which is built on the Extension Protocol, by decreasing the chances of peers exchanging contact information with each other. The peer exchange only takes place between two peers both having the protocol enabled, however accidentally reserving slots for peers that do not support/disabled the protocol decreases the efficiency by wasting the available slots intuitively. With the current fix, the PEX candidates are selected only from the peers that surely have their PEX protocol enabled.
- The libTorrent client removes a piece from its interest list as soon as a block of the piece is requested for download. However this early removal of a piece from the interest list can be problematic if the block transfer is cancelled due to departure of the remote peer. During the experiments, some peers have been noticed to get stuck with having 99% of the content and only missing a single piece to complete the download. The problem of getting stuck was caused by the wrong policy of the libTorrent, where a piece was removed from the interest list without completely being received. When a block transfer is cancelled, the libTorrent client fails to re-request it from other peers, despite having it available in its neighbourhood, as the piece is no more marked as interested. The peers getting stuck at 99% can be considered as an altruistic seed remaining in the system after completion, making it harder to define a fixed seed/leecher ratio during the experiment thus causing deviations from the expected results. In order to overcome this problem, firstly the interest removal policy has been changed such that a piece is considered as not needed if and only if all the blocks of the particular piece have been completely transferred. Secondly, if a transfer is cancelled the piece is immediately re-requested if it is available in the neighbourhood. It should be noted that a block is still requested only from a single peer at a time except the end game mode.

Applying the modifications above, rTorrent becomes a suitable candidate to be used in the experiments. Considering the low resource consumption of the client, an average computer in recent days can run approximately hundred clients concurrently without observing dramatic performance loss; however the number of clients running in parallel still affects the overall system performance.

Besides libTorrent, Mainline and Transmission clients were also considered appropriate for the experiments. Unfortunately the Mainline client, which is written in Python, has been eliminated due to seldom failures on tracker connection. Despite being an appropriate client for the experiments, Transmission is selected as the backup nominee due to previous experience on libTorrent.

4.2. Selecting BitTorrent Tracker

In order to perform BitTorrent experiments, a BitTorrent tracker should be deployed on the framework. As the tracker behaviour is not the main concern of this thesis, any application performing the default tasks of a BitTorrent tracker can be used. opentracker [23], developed by Dirk Engling, which is an open source and free BitTorrent tracker is used during the experiments. The tracker related parameters such as announce interval and number of peers in a tracker reply can simply be adjusted through the easy to use configuration file of opentracker.

4.3. Experimental Setup

In order to evaluate the algorithms described in this work, a private testbed for performing content distribution experiments has been developed. The experiments could also be performed by connecting a public torrent with a single peer; however it would not be possible to collect measurements about overall performance of the system through the single peer. Besides allowing extensive measurements per peer and overall system, the private testbed provides a fully controlled environment for experimentation thus facilitating evaluation of the system under artificial scenarios.

The rest of this section is organized as follows. First, important properties of the framework that enable faster and reliable experiments are mentioned. Later the boundaries on system resources and possible methods to relieve them are discussed.

4.3.1. Running Experiments in Steady State

In steady state a system has numerous properties that are unchanging in time. For swarming systems, steady state can refer to the state where the torrent size doesn't change radically with the evolving time. While peers are arriving and departing, the number of leechers and seeds hence the ratio of seeds and leechers (S/L ratio) remain constant if the swarm is in steady state. Hence running experiments in steady state provides the opportunity predefine the swarm size and S/L ratio.

The Little's Law can be applied to the swarming systems if they are in steady state. According to the theorem, the long-term average number of leechers in a stable swarm $l(t)$ is equal to the long-term average peer arrival rate, λ, multiplied by the long-term average of download completion time, T, or expressing algebraically:

$$l(t) = \lambda T \qquad (1)$$

The theorem can also be applied to the seeders if leechers do not leave the swarm without completely receiving the file and stay for some time altruistically after download completion. Let $s(t)$ to represent seeders in the swarm and Δ for average seeding time of seeds then the below equation also holds.

$$s(t) = \lambda \Delta \qquad (2)$$

In order to estimate the download completion time, T, the average download rate of peers should be calculated. For the experiments the download rates of the peers are set as unlimited thus the system is only bounded by the available upload bandwidth. Let μ represent the uploading bandwidth of a peer then the maximum value of average download rate of peers, *d*, can be calculated as:

$$d = \frac{\mu * \bar{x}}{\bar{l}} \tag{3}$$

Given file size and upload rate, the average download completion time, T, can be calculated as d can be calculated. The estimation gives the lower bound for the download time since it assumes that all peers fully utilize their upload bandwidths.

Before starting an experiment the number of leechers and seeders, consequently S/L ratio, are determined by the users. In addition to that if the upload rate of peers are given, the download rate hence average download time of peers can be estimated using (3). Finally the required parameter for the experiments, mean arrival rate of peers λ, can be found using (1). The average holding time of the seeds, Δ, is determined according to total download time and desired S/L ratio.

As the parameters are determined the initialization step bootstraps the swarm from the steady state. The peers that start up the swarm, namely the initial peers, are started as seeds or leechers according to the given S/L ratio. In addition to that the initial leechers start as some share of the content already downloaded. The fraction of content that each leecher will start with is generated uniformly random. Also the pieces to be possessed by each peer are picked uniformly. Therefore the leechers start with possessing half of the content on average and the pieces in swarm are distributed uniformly thus equally available. It should be highlighted that in steady state of a BitTorrent swarm, the overall average content availability of peers should be close to 50% and the availability of pieces should be equal as aimed by the rarest-first piece selection policy.

A remarkable result that can be derived from Little's Law is that the peer arrivals are entirely independent of its distribution but simply mean arrival rate is considered. Therefore various arrival patterns can be used for the experiments as long as the mean arrival rate is kept constant. By default the peer arrivals are generated according to the Poisson process. However sometimes it can be tough to find the suitable peer arrival rate that keeps the system in steady state. In order to find a good estimation of arrival rate the arrival-per-departure pattern has been used. In arrival-per-departure method, a new peer joins the system only if another peer leaves the system thus the swarm size is always kept constant. It should be noted that currently arrival-per-departure pattern can only be used if S/L ratio is set to 0 or in other words the leechers leave the swarm as soon as download completes. However it can be modified to force a seed for departure and start a new peer when a leecher finishes downloading in order to keep ratio of seeds and leechers constant.

According to the results collected with arrival-per-departure pattern, a better estimation for peer arrival rate can be determined rather than the value calculated using Little's Law. Later the estimated value can be applied to Poisson process for generating peer arrivals. *Figure 9* shows the distribution of peer inter-arrival times when arrival-per-departure pattern is used. Fortunately the inter-arrival times are distributed exponentially same as the Poisson process. The peer inter-arrival times' distribution of swarms with 60 and 150 peers fit well with exponential distribution with mean values 22.3 and 8.75 respectively. The generated values also hold when applied to Equation (1) if download time is kept constant.

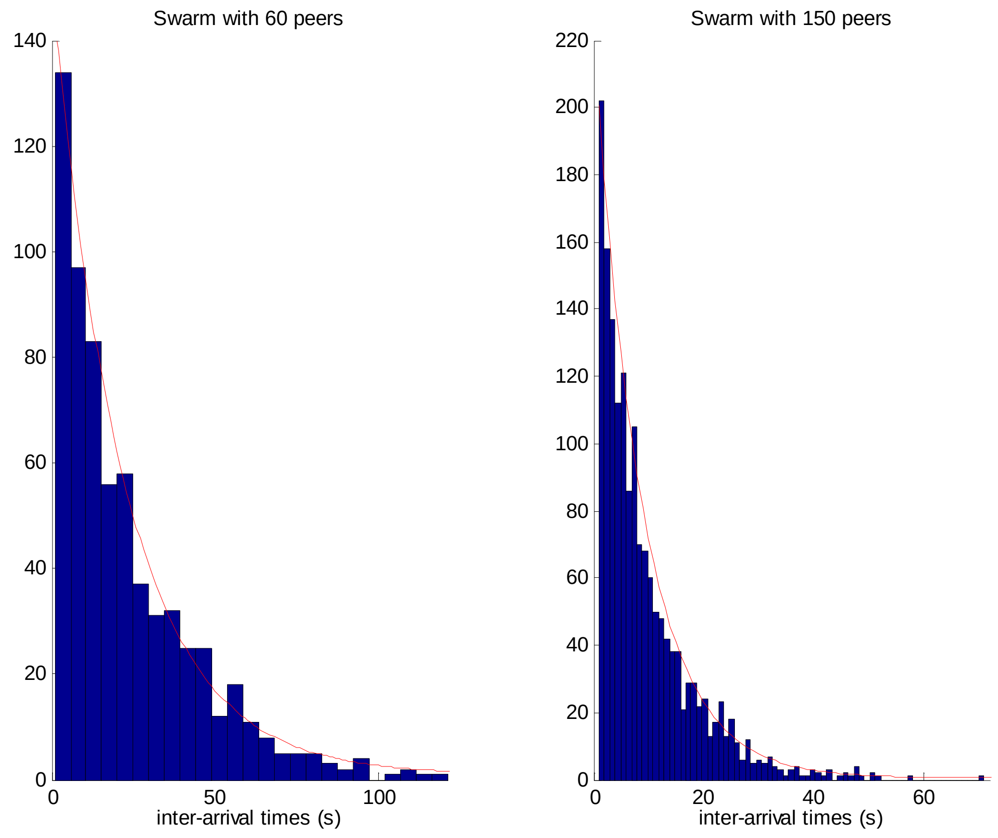
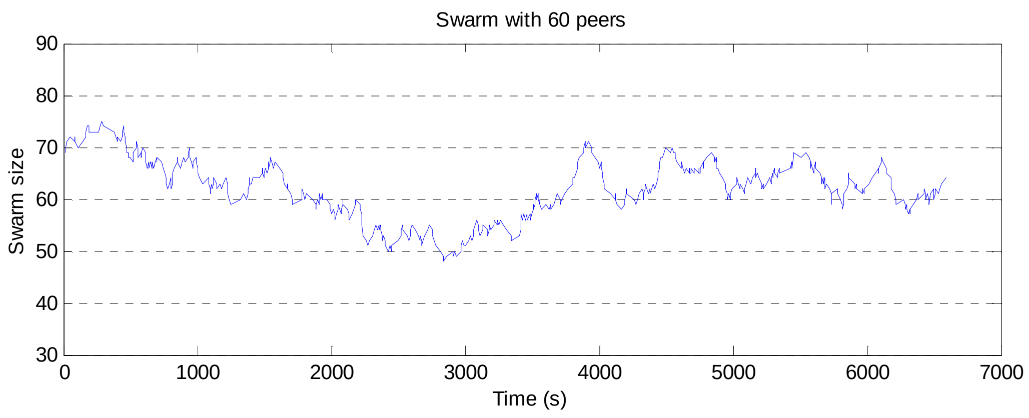


Figure 9: Peer inter-arrival times distribution with arrival-per-departure pattern

Upon download completion the leechers may stay in the swarm to help others if seeding time value is assigned by the user. The assigned seeding times can be generated exponentially or with normal distribution according to the user defined mean value for seeding time. In most of the experiments performed, the peers leave the swarm as soon as they completely receive the file, namely selfish peers are preferred for the sake of simplicity.



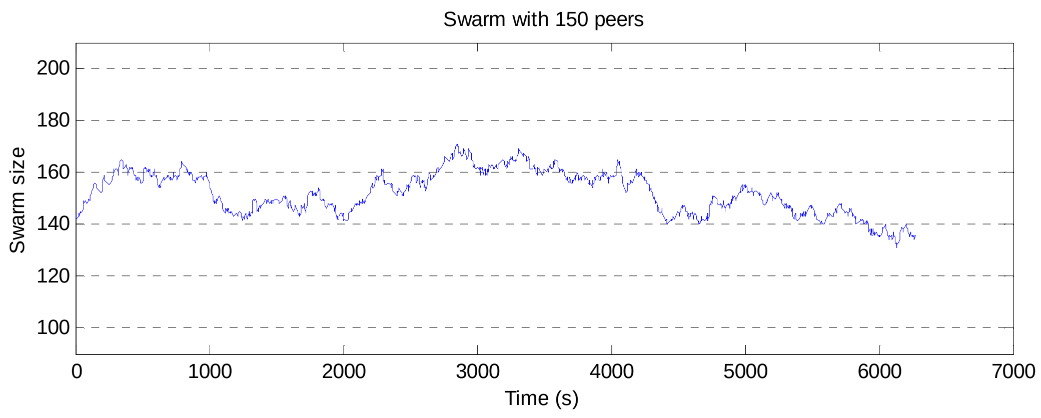


Figure 10: Swarm size vs. time plot representing steady state behaviour

Figure 10 represents the stability of the swarm size in two separate experiments with swarm size 60 and 150 with peer arrivals according to Poisson process. The swarm sizes never show a dramatic change during the experiment hence implying the steady state behaviour of the system. In addition to that the mean values of swarm sizes are calculated as 62.1 and 152.3 for the experiments plotted above which are very close to the user defined value.

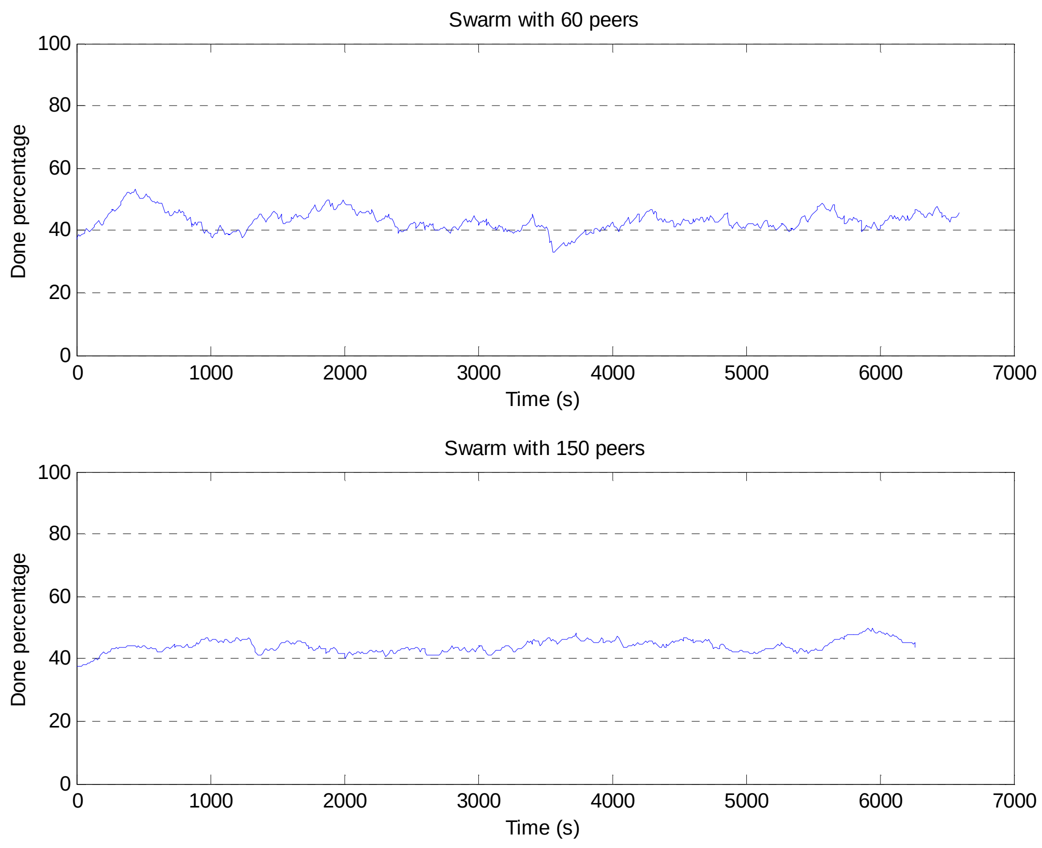


Figure 11: Done percentage vs. time representing steady state behaviour

Figure 11 shows the average of download completion percentages as a function of time. As can be observed in the figure, the overall average content availability flickers between 40% and 60%; representing the steady state characteristics of the system.

4.3.2. Downscaling Experiments

In recent days BitTorrent is mostly used for downloading files with size 600MB or more such as movies or TV episodes. Furthermore high definition video formats that gained popularity recently increase the size of the downloaded contents. Depending on the available bandwidth of users and swarm dynamics, the download completion time can vary from 10 minutes to 2 hours where with a rough estimation can be rounded as an hour on the average. Considering the default BitTorrent settings used by most of the clients, the peer discovery is performed by tracker queries every 30 minutes and peer exchanges every 2 minutes.

Even though achieving a highly realistic test environment is aimed, it is not feasible to perform experiments with 600MB file downloaded in an hour. First of all, the high content size dramatically decreases the number of clients that can be run simultaneously on a single machine. In addition to that the long download time causes a single experiment to take around 5 hours. Since the experiments contain stochastic behaviour such as random peer arrivals, they need to be repeated multiple times to increase the confidence of the collected statistics. If each experiment is repeated 10 times then experiments with single parameter change takes around 50 hours which is not feasible. Therefore the experiments needed to be downscaled while preserving the important properties of the system.

To increase the number of peers that can be run on a single machine the content size has been set to 50MB. The download time has been decreased to 10-20 minutes which is considerably long enough to observe swarm dynamics and feasible even the experiments are repeated. In order to preserve the ratio between sources of peer discovery, tracker query intervals are set to 1 minute and peer exchanges are done every 30 seconds. To be able to clearly observe the effects of peer exchange, the tracker replies can be limited to contain only half number of peers of the connection list size. With the current settings a peer queries the tracker approximately 10 times during its lifetime but actually only the first query is important since peer exchange successfully fulfils the remaining slots until the following tracker query in most of the scenarios.

4.3.3. Providing Global View to Initial Seed

Despite having very low probability it is still possible to form a cluster in which none of the peers in this subset have the initial seed in their connection list. Such a scenario can end up with a group of peers getting stuck due to unavailability of some pieces. In order to avoid such scenarios the publisher is set to stay forever in the torrent with a global knowledge of the swarm.

Different from other peers the publisher does not have limitations on size of the connection list hence is allowed to connect to as many peers as possible. In addition to that the tracker behaviour is slightly modified so that the tracker replies always contain the address of publishers on top of the list. The publisher can be identified in different ways such as assuming the first peer connecting to tracker as the publisher or reserving a well-known IP address for the publisher. Since the publisher is allowed to accept any incoming connection every peer is guaranteed to have at least one source for all the pieces.

Despite not being able to serve all peers at the same time, the publisher guarantees aliveness of the swarm. If a peer fails to get a piece from all of its neighbours the piece will eventually be leeched from the publisher upon an optimistic unchoke.

With having global knowledge of the torrent, the publisher offers a chance to observe swarm dynamics at run time. Depending on the properties provided by the client’s user interface, users can gather quick insight about the experiment only by the information shown by the publisher. The evolvement of some torrent properties can easily be obtained by logging publisher information with regular intervals. However current implementation of the framework processes log information of each peer separately after the experiment finishes. To obtain a global knowledge the data obtained from all peers are combined and processed. Despite being costly this way of information gathering is preferred to obtain more reliable test results.

4.3.4. Determining Warm-up Period and Experiment Length

The reliable output data analysis requires correct determination of warm-up period and measurement length. The warm up period is the time interval spent until the system achieves its steady state. Correct determination of warm up period prevents ending up with results biased by initial conditions. The challenge in identifying a suitable warm up period is that it should be long enough to avoid effects of initial settings yet not so long as to avoid excessive waste of collected data.

The longer measurement of experiments helps reducing the effects of stochastic behaviour hence increases the confidence of the results. Besides, longer runs help removing the effects of initial settings since the weight of the steady state increases as the experiment continues. However a suitable length has to be determined to perform the experiments in a reasonable amount of time.

As mentioned above current testbed bootstrap the experiments from the steady state. Therefore no initial data has to be ignored to avoid effects of initial settings. However since it is not possible to precisely measure the properties of initial peers, the peers that start with some share of the content as already downloaded, the data collection starts after all initial peers leave the system. In other words the system is warmed up until the founders of the swarm leave. It should be noted that the number of peers that start the swarm tends to remain stable as the system is in steady state.

In order to perform reliable analysis it is important to collect sufficient data within the shortest possible time. To determine the suitable measurement length some experiments are performed for very long duration and the point where experiments converge is selected as the termination time. The number of peers departing from the swarm is selected as time metric to have a common unit with the warm-up period. To define a generic metric for experiments with varying swarm sizes, the time unit is normalized with the swarm size so the time is measured as ratio of departing peers to the swarm size.

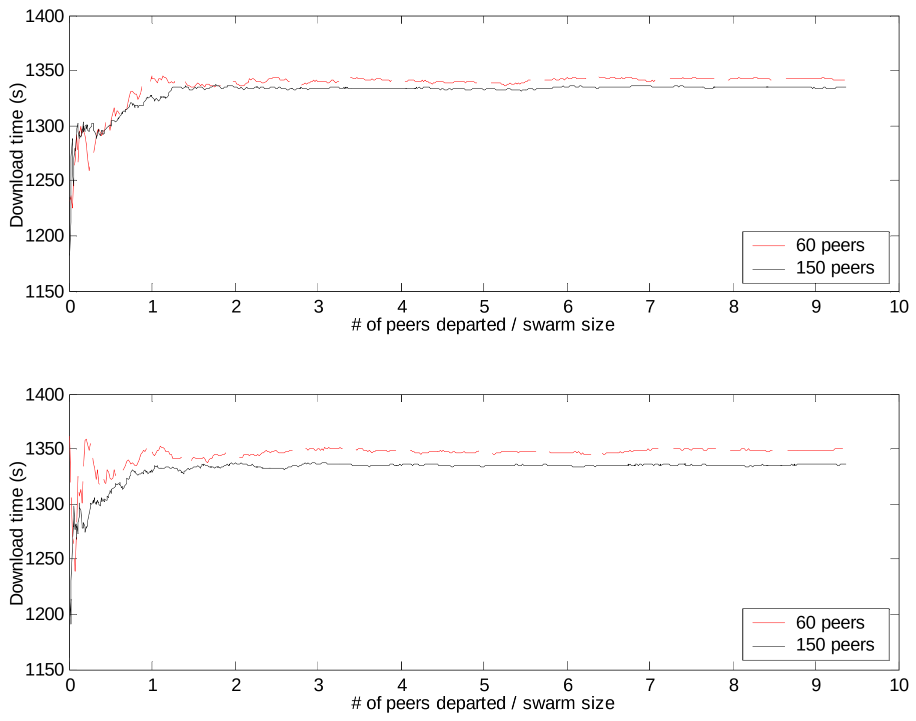


Figure 12: Two long experiments for measurement length determination

Figure 12 represents the results of experiments performed to determine the suitable experiment length. Two experiments with swarm sizes of 60 and 150 peers are run until 10 times the swarm size, 600 and 1500 respectively, peers depart from the swarm. Each line represents different runs of the experiments. As can be seen from the figure, the download time average converges after 3 times the swarm size peers depart. Therefore the experiments are run until 3 times swarm size peers leave the system after the warm up period. For more reliability the experiments can be run until 4 swarms depart however the results of them would not differ.

4.3.5. Emulating Real Network Characteristics

In order to evaluate the BitTorrent system with its realistic environment, a network emulator named ModelNet [21] has been deployed on the computers. When provided a network topology and routing information, ModelNet enables testing of distributed systems across various networking scenarios. Within a private network ModelNet can be used to create a high number of virtual nodes to emulate large-scaled networks with realistic characteristics. The emulator can introduce network latency between two virtual nodes running on the same machine or machines in the same local area network (LAN). Consequently, the BitTorrent behaviour on large-scale systems can simply be evaluated under realistic network characteristics in a small private LAN by running a single peer on each virtual node.

On the contrary in their work [22], the authors measure the impact of network latency on download completion time in BitTorrent. According to their findings the network latency has a marginal effect on the download time, which is less than 15%; hence the necessity of network emulation in a testbed can be relieved.

The developed testbed allows enabling and disabling the network emulation by setting a parameter in the configuration file. According to experiments in developed environment, the results with and without network emulation did not seem to differ much as mentioned in [22]. In addition to that current work mainly focuses on increasing connectivity among swarms rather than throughput measurements. Therefore ModelNet emulator has been set as disabled during the rest of this work.

4.3.6. Boundaries on System Resources

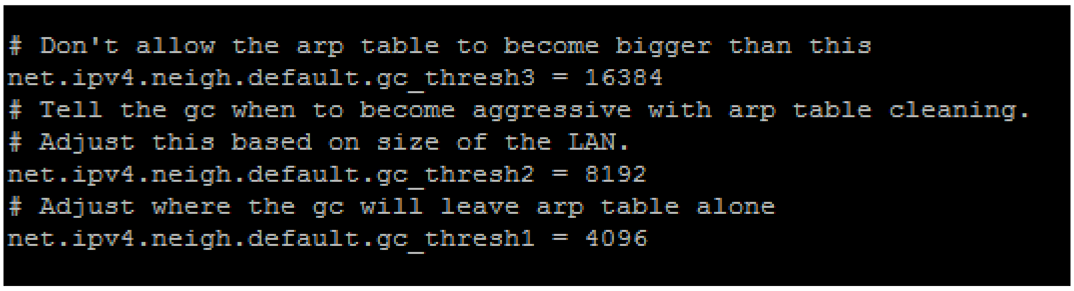
The developed testbed aims to allow running experiments with as many peers as possible however it is bounded with system resources. The testbed is currently deployed on a dedicated platform consisting of 6 computers in which one of them is reserved as the master computer to control the execution of the experiments and the rest are used as slaves for running BitTorrent clients. The master computer has quad-core Intel Xeon 2.83GHz CPU and 4GB memory. The slave machines are equipped with dual-core Intel Xeon 3.40 GHz CPU and 8GB memory. The computers are connected to each other through 100Mbit/s Fast Ethernet. The computers are used for running a single experiment at a time to maximize the scale of the experiments.

The boundaries limiting the maximum number of client instances that can be used in the experiments are discussed as follows. Also some key modifications that help relieving these limitations are mentioned.

- The delays caused by hard disk I/O waits increase as the number of clients running on a single machine increases. To avoid performance degradation due to disk delays with increasing number of clients, a partition has been mounted to the memory for client use. Although this approach removes the boundary caused by excessive disk access delays, the maximum number of clients gets limited by the available memory which is smaller than the disk drive. To efficiently use the limited RAM disk, the space used by a client is de-allocated upon departure hence only the number of clients that can be run simultaneously is bounded. The current settings reserve 4GB of available 8GB memory for the RAM disk hence 60 clients can simultaneously download a content of 50MB without available space or I/O boundaries. A space of 1GB is reserved for the proper functioning of the framework.
- The available network bandwidth is another resource that limits the number of clients running on machines. In order to determine limitations of network resources, the bandwidth requirement of the experiments are increased gradually, either by varying number of clients or the rate limitations, and the network utilization of computers are monitored by the *iftop* tool. The utilization measured by the *iftop* tool increased as expected until the bandwidth requirement reached around 30Mbit/s. The experiment parameters are set accordingly so that the aggregated upload capacity per machine never goes beyond 30Mbit/s, for instance 60 clients per machine with 40KB/s rate limitation or 30 clients per machine with 80KB/s requiring 19.2Mbit/s per machine, to ensure that all of the clients can achieve the maximum rates they are assigned without running out of available bandwidth. For experiments that mainly aim to measure the relationship among swarms, the required bandwidth can be

doubled, 60 clients with 80KB/s rate limit requiring 38.4Mbit/s at total, to half the duration of the experiments without badly harming the download performance.

- Basically the address resolution protocol (ARP) is used for determining a network host’s hardware address when only its IP address is known. ARP is most frequently used for translating IPv4 addresses into Ethernet MAC addresses therefore it is crucial for transmission in local area networks. By default in Linux kernel implementations the ARP cache is limited to store 512 entries. However for networks with more than 512 nodes it is mandatory to increase the kernel’s internal ARP cache size to avoid failures on packet delivery due to table overflow. To overcome this problem /etc/sysctl.conf file in all machines has been edited to allow up to 16384 entries in the ARP cache which is quite sufficient for the experiments. Also cache size thresholds are updated accordingly to avoid excessive calls for garbage collection. The updated settings can be observed in *Figure 13*.



```
# Don't allow the arp table to become bigger than this
net.ipv4.neigh.default.gc_thresh3 = 16384
# Tell the gc when to become aggressive with arp table cleaning.
# Adjust this based on size of the LAN.
net.ipv4.neigh.default.gc_thresh2 = 8192
# Adjust where the gc will leave arp table alone
net.ipv4.neigh.default.gc_thresh1 = 4096
```

Figure 13: ARP cache parameters in /etc/sysctl.conf file

- Upon download completion the client performs hash check to whole file which heavily consumes the CPU and I/O resources. The delay caused by final hash check increases exponentially if the number of clients goes above a certain limit. To overcome this situation final hash checking is disabled by adjusting client parameters. Fortunately hash checks are still performed upon piece completion without causing excessive delays.

Although it is obvious it should be re-stated that increasing the number of clients per machine causes a notable performance decrease on the system due to increasing consumption of resources.

4.4. System Components

The goal of this section is to briefly discuss the functionality of the deployed components and to give insight to the reader about how the system works. The section is organized according to execution order of components during the experiments.

4.4.1. Initializing Computers

The testbed requires at least 2 computers to function properly however there is no upper limit on the number of computers that can be used for the experiments. One of the computers is reserved as the *master* to run trackers and original seed (publisher), to control the experiments and to perform analysis of the log files. The remaining computers are used as *slaves* and are used to run the peers for the experiments. In order to register a slave computer to the testbed, the configuration file on the master computer should be edited by adding the IP address or alias of the new computer to the list of slaves.

It should be noted that it is *not* mandatory to use all the registered slaves for an experiment. A subset of available slaves can be assigned for an experiment. This flexibility provides the chance to perform the experiments under equal load, running the same number of peers on each computer. For instance if the load is set as 30 peers per machine, torrents with 30, 60, 90, 120 and 150 peers uses 1, 2, 3, 4 and 5 slaves respectively. For the rest of the report, the word *slaves* can be used interchangeably either to represent all the slaves registered to the testbed or the slaves reserved for the experiment when it is clear from the context.

All communication among the computers is performed using the SSH and SCP tools. To avoid automated scripts from getting blocked, the password enquiry of protocols should be removed. In order to do so, a public-private key pair has to be generated and located under *.ssh* directory on all machines. Having same username and password on all computers, any machine in testbed can invoke SSH and SCP on every other without entering the password manually. To avoid establishing a new SSH connection for each remote command execution, connections from master computer to all slaves are created and stored in background using the *screen* command as long as computers keep functioning properly. The remote calls through the connections in background are asynchronous; no clue is given about the termination of the command since the return values are not collected from the background connection. When a command block is required to execute sequentially, the commands should be invoked synchronously by creating a new connection for the particular set of commands. The SSH tool blocks the execution until the issued command terminates if the protocol is used with the syntax below:

ssh machine_address "remote command"

However asynchronous calls are performed by simply typing the command through the connections running in background and continuing execution without waiting the result of the command.

As mentioned in earlier sections the clients make use of RAM disk to avoid I/O delays due to slowness of hard disk drive. Therefore a script that mounts the RAM disk on all computers has to be started before using the computers. The script executes the command below on all slaves and the master itself:

sudo mount –t tmpfs none /var/ramdisk –o size=${1}m

The script requires the size of the RAM disk in megabytes as a parameter since available hardware on machines can be different. The RAM disk is volatile and is removed when the computers are restarted.

In order to avoid any IP address conflicts, each peer is assigned a unique IP address and port pair. Some clients are capable of running multiple instances on same IP address when different ports are assigned however there still can be conflicts on clients or trackers. In order to make it secure a unique IP address is reserved for each peer by dynamically creating a new ethernet interface using the *ifconfig* command. However this task requires super user privilege thus requires a password to be entered manually. To automate this task the line

username ALL=NOPASSWD:/sbin/ifconfig

has been added to the "/etc/sudoers" file on all machines so that the command no longer requires authentication. Although modifying the privileges of every user can be considered as not secure, this modification can be considered harmless for the current deployment as the computers are behind a firewall.

The files to distribute is stored under the directory defined by the shell variable *CONTENTS*. Any content can be used for experimentation, however due to legal issues the testbed currently uses text files of various sizes containing random printed characters. The meta-info files containing information about the torrent, such as tracker announce addresses, should be created manually and placed under the same directory with the content. The *mktorrent* application under Linux can be used for meta-info file creation. The *CONTENTS* directory should be replicated on every slave. It is preferable to locate this directory on RAM disk for faster access.

Each client is assigned a separate directory under the *PEERS* directory which is also located on RAM disk. The clients use the assigned directories for session management, storing the downloaded file and logging down statistics.

Finally a directory called *RUNS* has to be present on the master computer to store the processed results of the experiments.

Each computer should possess the applications and scripts required for testbed execution. In addition to that the client application should be installed on all computers and the master computer should be able to start a tracker application.

4.4.2. Initializing Experiments

The initialization script is an automated tool to make the experiments ready for bootstrapping according to the given parameters. As mentioned, each client is assigned a workspace under the *PEERS* directory. The client workspace is organized in a structure formed by *FILES, SESSION* and *STATS* directories which are used for storing content, keeping session data and logging statistics respectively.

The number of trackers, publishers and peers in the torrent is determined by the user prior to the experiments. The whole content is copied under the directories of publishers running on the master computer. If swarms are expected to have equal number of publishers, meta-info files containing the address of swarms one-by-one should be used to register the publishers in different swarms. An alternative way used for fair seed power distribution is running a single publisher that registers to all existing swarms using a modified client.

As mentioned earlier, the initial peers start with some content as already downloaded in order to bootstrap the system from steady state. An application is executed on the master computer in order to assign the clients to run on the reserved slaves while distributing the workload equally on the slave computers. The program also calculates the percentage of content that each peer will start with as downloaded already. A temporary script is generated for each slave that initializes the peers assigned to that particular slave. Running the temporary script, slaves are notified through SSH to initialize the peers assigned to itself. The application *createContentWithGivenPieceNumber* is executed by slaves to generate partly downloaded contents where the downloaded fraction for each peer is already determined by the master computer. However the pieces assigned to each peer are picked randomly by the file generator. Additionally the Ethernet interfaces are created on the slaves during peer initialization to reserve a unique IP address for each peer. A random value for time to seed is generated for each peer if peers are set to continue seeding after download completion.

The execution in the master computer is blocked until the peer initialization on slaves is finished to ensure that experiments do not start before the initialization phase completes successfully. The initialization phase results in a swarm in steady state, in compliance with the parameters given by the user in terms of torrent size and S/L ratio.

4.4.3. Running Experiments

As soon as the initialization phase is accomplished, the script for starting the experiment can be invoked. Depending on the number of swarms defined by the user, trackers are started with different announce addresses on the master computer. At least one publisher is available to each swarm however this number can be adjusted by the user. The publishers are also started on the master computer. As mentioned earlier the testbed allows publishers to have the global knowledge of the torrent therefore the connection list size of publishers are set as infinite. Having contact information of every peer in the torrent, the publishers can have a very high impact on mixing of the swarms which can hide the actual performance of the algorithms. Therefore the publishers are not allowed to perform peer exchanges. The remaining settings of publishers are kept same with other peers.

As soon as publishers are started, the slave machines are instructed to start the initial peers. A virtual terminal is created with *screen* command for each peer. To avoid address conflicts each peer is assigned a unique IP and port pair. The settings file of the libTorrent client, *rtorrent.rc*, is distributed on all machines with same parameter values. However some parameters such as upload and download rates, connection list size, number of simultaneous uploads, time to seed etc. can easily be overwritten via the testbed settings file. This property facilitates running experiments with heterogeneous peers. The script first types the commands for starting each client in the terminal screens then all clients are started simultaneously by just sending the *enter* key to screens to execute the typed commands. By means of this approach it takes no more than 10 seconds to have all the peers joined to the system.

In order to control the experiments the *controller* application runs on the master computer and the *reporters* run on the slaves. These tools continuously collect information about the experiment status and control the flow of the experiment. The *controller* establishes a TCP connection with each *reporter* for communication purposes. As known TCP connections provide communication on both directions. Peers notify the testbed upon 3 important events:

- Joining a swarm
- Download completion
- Leaving the torrent

The peers communicate with the *reporter* running on the same machine via named pipes (also called FIFO). A named pipe is a special type of file that is used for inter-process communication. Using pipes the output of one process can be used as input to another process. The reason for using named pipes instead of regular files is that the file descriptors referring to pipes allow synchronous I/O multiplexing using the Linux *select()* function. With the *select()* function an application can wait and monitor a set of file descriptors until a change occurs in the monitored descriptors. Considering the testbed, when a peer performs one of the mentioned important events it executes a shell script to notify the controller about the event. The script simply writes the event parameters to the named pipe created on the slave computer. Due to the change in the pipe, the reporter gets notified and receives the event. The reporter simply redirects this message to the controller over TCP sockets. The peer's message is processed by the controller and state variables of the experiment are updated accordingly.

Throughout the experiment controller may decide to manipulate the swarm. For instance a new peer can be added to the torrent or the experiment can be terminated. These commands are sent to reporter to be executed on the slaves.

As mentioned earlier two different arrival patterns, Poisson process and arrival-per-departure, can be used. In order to generate peer arrivals with Poisson process a separate process is forked by the controller after all initial peers join the torrent. The Poisson arrival process creates exponentially distributed peer inter-arrival times according to the user given mean arrival rate. The arrival messages are transferred to the controller through the existing named pipe. In arrival-per-departure pattern the controller itself decides to start a new peer upon receiving a peer departure event. To start a new peer, the controller picks the slave with lowest load and transmits a peer start event to the corresponding reporter. Upon receiving the command, the reporter invokes the shell script that initializes and starts the new peer.

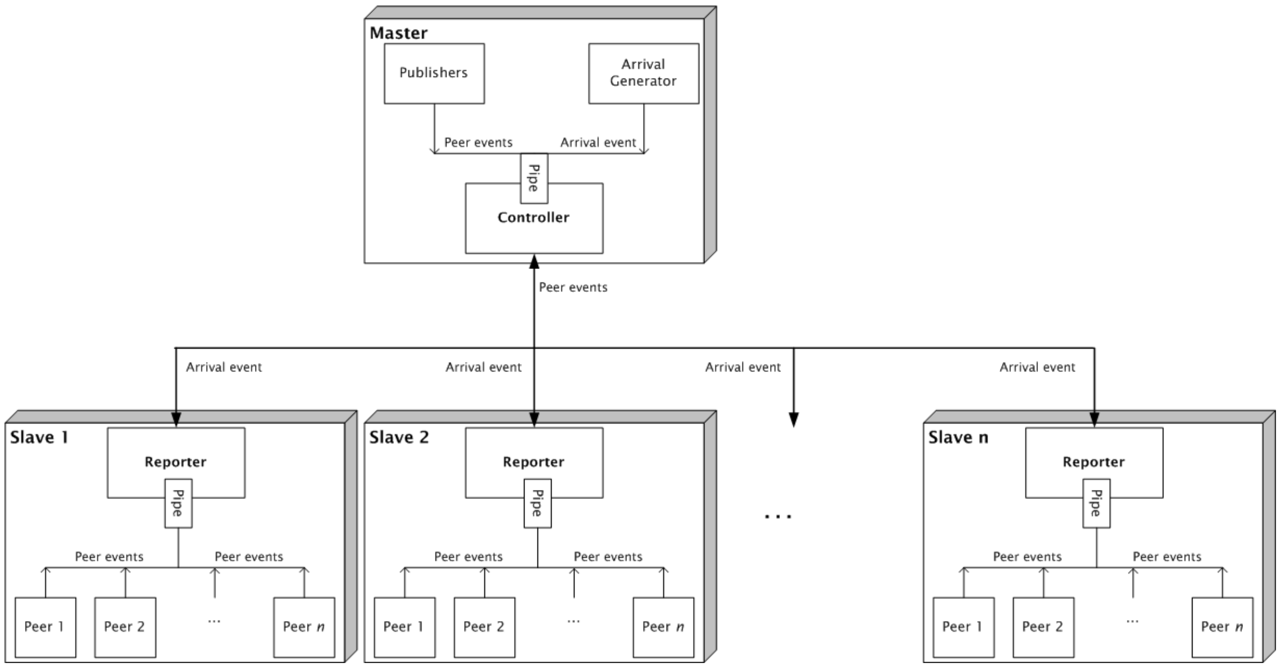


Figure 14: Communications between components

Figure 14 represents the communications and delivered events between the components. As can be seen from the figure the peers notify the reporters through unidirectional channels implemented by named pipes. The peer events are transferred from the reporters to the controller via the TCP connection between them. The controller can also send commands to the reporters through the same channels.

To avoid waste of disk space, the reporters remove the content downloaded by peers upon departure therefore the disk space allocated by peers can be kept constant through the experiment.

In order to have a globally synchronized clock, the controller measures the drift between itself and slaves by calculating the difference between timestamps of the first messages received. The time difference between the master and each slave is stored in a table and all incoming messages from slaves are updated according to the difference of that particular slave. Using this time synchronization method, it is possible to achieve loose synchronization during the experiments at a low overhead. Loose synchronization is sufficient for our purposes, as a drift of one second can be tolerated in the statistical calculations.

If realistic network emulation with ModelNet is enabled, clients and trackers are executed in encapsulation of *vnrunhost* processes. Each peer and tracker is assigned a unique id in ModelNet where each id represents a virtual node in the user defined network topology.

The controller provides an interface to the user to gather information about an ongoing experiment. The user can watch the current state of the torrent and observe the calculated statistics till that time. Also users can terminate the experiments earlier when necessary through this interface.

4.4.4. Terminating Experiments

The experiments are terminated after the experiment length determined by the user passes. The experiment length is currently defined by the number of departed peers over swarm size thus providing the opportunity to measure experiments with different sizes of swarms for equal time. In measurements performed the experiments are shown to converge after three generations of peers depart hence departure of three swarms is set as the default parameter for experiment length. Another alternative for experiment length is setting a measurement time interval in seconds. Lastly, the user can terminate the experiment manually through the controller's user interface.

Once controller decides to terminate the experiment, all inter-process communications towards the controller are closed to ignore any other incoming messages. Later a script for terminating all relevant applications on slave computers is executed. This script kills the reporters and clients running on the slaves.

Before terminating itself, the controller fetches all peer logs from slaves and stores them together in the master computer. The controller processes the collected log files to calculate the results of the experiments. The processing of the log files will be discussed in more detail in the following section. As the output generation completes the controller terminates and the script for killing the remaining applications such as trackers and arrival generators is executed.

4.4.5. Processing Collected Data

During the experiments each peer records the time of joining, download completion and departure. In addition to that all events performed on trackers such as registering or leaving a swarm, announces and scrapes are logged for load measurement on trackers. Furthermore peers periodically, at every 10 seconds, log the addresses in their connection list so that connectivity relationship among swarms can be observed.

Upon experiment termination the controller fetches log files on slave machines recorded by the peers. For each peer, the log files are matched with the event messages received during runtime to verify the correctness of the system. If a mismatch is detected the user is warned and the data in log files are used as they are more reliable.

The logs of each peer are processed separately at first. Given the joining, completion and departure times of a peer the download and seeding times can be calculated easily. It should be noted that the time values in log files are local to the slave machines thus they are adjusted according to the drift between computers.

All peer events are inserted to a global queue and the events are processed according to chronological order. In other words the experiment is re-traced through events from log files to generate output from the collected data. Only the events relevant to the results of interest are recorded. While simulating the experiment the state information maintained by the controller is updated according to the events. The mentioned events and state variable maintenance are as follows:

- *Joining to the torrent*: When a peer join event is received, a new peer object is created and inserted as a leecher.
- *Download completion*: Upon download completion of a peer, the peer is removed from leechers and inserted as a seed.
- *Leaving the torrent*: The peer is deleted from the seeds. It should be noted that peers are not allowed to abort download and leave the torrent in current implementation thus any departing peer is guaranteed to be a seed.
- *Registering to a tracker*: The peer object gets associated with the given swarm. A register event is the first announce sent to the particular tracker.
- *Unregistering from a tracker*: The peer object is removed from its swarm. A peer may unregister either to join another swarm or leave the torrent.
- *Announcing to a tracker*: Announces are only processed to measure the load on trackers.
- *Scraping a tracker*: Scrapes are special types of announces such that the tracker returns number of peers and seeds in the swarm and number of times the content has been snatched

instead of returning a set of active peers. Despite being cheaper than announces scrapes should also be considered as load on a tracker.

- *Update connection list*: The peers log down the addresses of peers in their neighbourhood every 10 seconds thus connection list updates are triggered periodically by each peer. Upon this event the neighbourhood of the peer is simply replaced with the new set of peers.
- *Calculate output variables*: *Calculate* events are dispatched every 10 seconds manually by the controller. When this artificial event is triggered, the global swarm dynamics are calculated. This can be considered as capturing instantaneous snapshots of the torrent in order to measure the dynamics through the whole experiment. The average of these parameters are calculated and presented to the user.

The events are processed from the start of the experiment however the *calculate* events are only scheduled after the warm-up period hence the warm-up period effects are avoided in measurements. The peer statistics measured always consider the peers after the warm-up period unless initial peer measurements are requested.

Various torrent statistics can be obtained through processing the above events. It is also possible to output the statistics by grouping the peers according to their swarms, the machines they run on or other distinguishing properties such as grouping the initial peers or special peers that exhibit mixing behaviour. Currently the measured properties are:

- Average number of leechers and seeds in torrent
- Peer arrival rate
- Download time and seeding time of peers
- Download rate and upload rate of peers
- Average node degree of peers (average size of connection list)
- Number of peers that completely received the file
- Total number of announce and scrapes performed by peers (overall communication overhead)
- Peer exchange traffic
- Connectivity among swarms
- Total number of peers running mixing algorithms
- Properties of peers running mixing algorithms
- Swarm based statistics: Average size of swarm, total number of scrapes and announces on the tracker
- Computer specific statistics: Overall performance of peers grouped according to the machines they run

In addition to that performance of every peer is printed to the output file. This file can be observed manually by users or processed by applications to mine meaningful data.

4.4.6. Resetting Experiment

In order to prepare the experiments for another run the testbed workspace has to be cleaned. A bash script is run by the master to reset all the experimental setup. The script first removes the directories used by the peers located under the *PEERS* directory. In addition to that the temporary files for global logging and communication purposes such as named pipes are removed. The script is also invoked synchronously by SSH tool to reset the slave computers. Different from the master, the reserved IP addresses are cleared on the slave computers using the *ifconfig* command. The reset script blocks the execution until all slaves are reset.

4.4.7. Automating Set of Experiments

The testbed can be used to run a set of experiments with different settings without user intervention. Users can place different settings files under test directories and run an automated script to perform the experiments. The number of repetitions for experiments can also be specified by the user.

The automated script simply combines the scripts explained above. Apart from the scripts above, the settings files are replaced at each experiment and the output of the experiment is saved under the corresponding experiment's folder with an assigned run number.

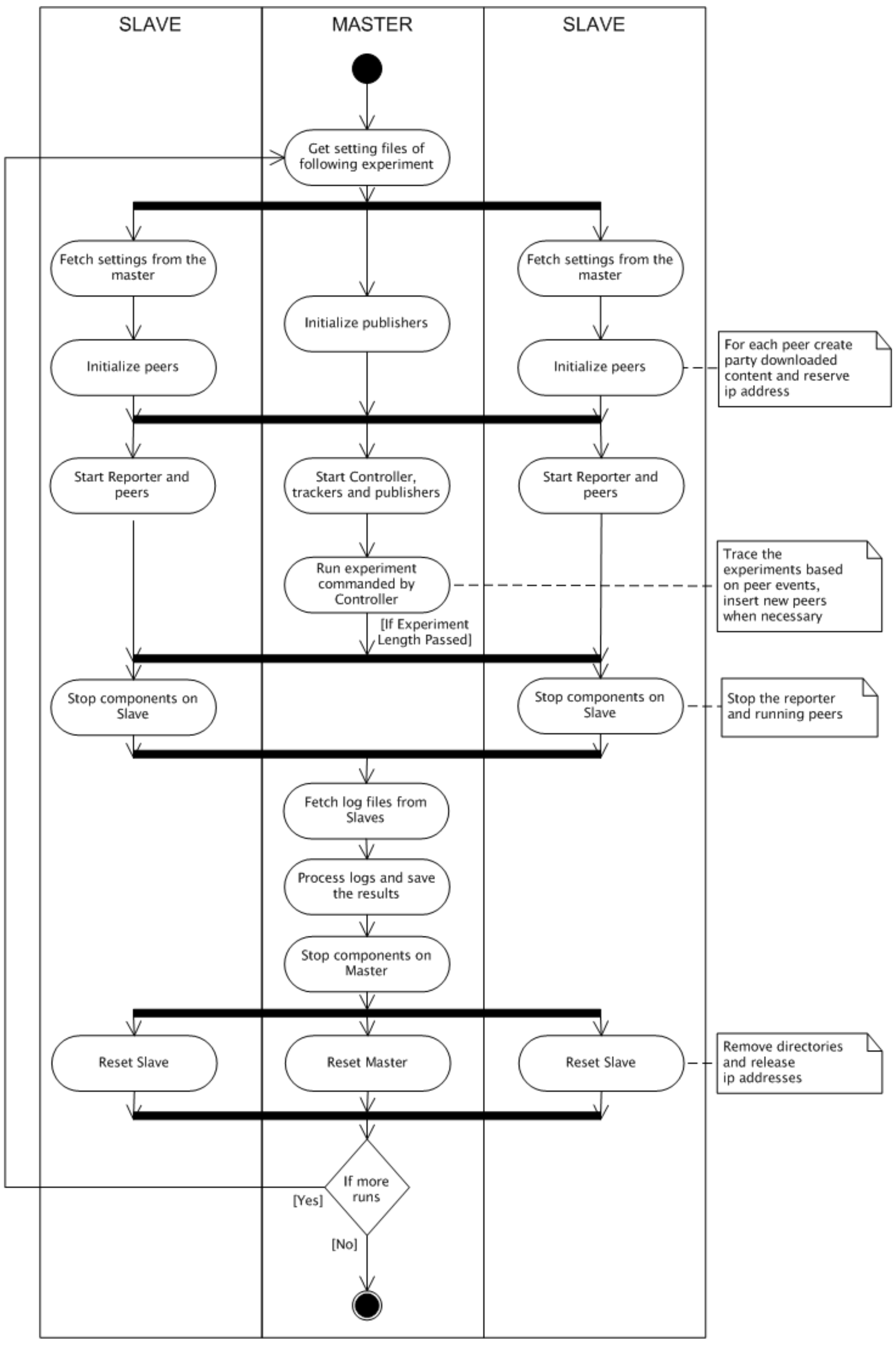

SLAVE
MASTER
SLAVE
Get setting files of following experiment
Fetch settings from the master
Initialize publishers
Fetch settings from the master
Initialize peers
Initialize peers
For each peer create party downloaded content and reserve ip address
Start Reporter and peers
Start Controller, trackers and publishers
Start Reporter and peers
Run experiment commanded by Controller
Trace the experiments based on peer events, insert new peers when necessary
[If Experiment Length Passed]
Stop components on Slave
Stop components on Slave
Stop the reporter and running peers
Fetch log files from Slaves
Process logs and save the results
Stop components on Master
Reset Slave
Reset Master
Reset Slave
Remove directories and release ip addresses
If more runs
[Yes]
[No]

Figure 15: Running multiple experiments with different settings automated

Figure 15 contains an activity diagram representing the execution order of the script that can be used for performing various experiments with multiple runs. The script successfully automates the task of running consecutive experiments by simply combining the tools mentioned above.

5. EVALUATION

This chapter presents an evaluation of the algorithms discussed in Section 3. The evaluation is done by analyzing the algorithms performance on the private testbed. First the metrics for quantifying performance of algorithms are described. Then the default settings for experiments are mentioned. Finally performed experiments and the obtained results are presented and discussed.

5.1. Evaluation Metrics

The effectiveness of the swarm management algorithms is quantified in terms of the following metrics:

- *Download Time*: The success of the algorithms is evaluated by their improvement in terms of overall download time. The mean download time of peers that join the torrent after warm-up period and completely receive the file are calculated.

- *Mixing Efficiency*: The performance of the algorithms is evaluated with a new metric, mixing efficiency, which is defined as the average number of external peers known by a particular swarm normalized with the torrent size. Mixing efficiency of swarm *r* is calculated as:

  $$M_r = \frac{x_r + \sum_{r' \in R \setminus \{r\}} \overline{y}_{r,r'}}{x}$$

  where $\overline{y}_{r,r'}$ represents the average number of external peers that are registered to swarm r' but known by swarm *r*. The average mixing efficiency of the overall system can be expressed as the weighted average:

  $$M = \frac{1}{x} \sum_{r \in R} x_r M_r$$

  It should be noted that without mixing algorithms no external peers exist in swarms, algebraically $\overline{y}_{r,r'} = 0$ for every swarm pair *r* and *r'*; however the mixing efficiency may not be computed as 0. As it is obvious the efficiency of mixing increases as mixing efficiency, *M*, gets closer to 1. The virtually increased size of each swarm can be estimated by reversing the process: $x * M_r$ gives an estimation of increased size of swarm r.

  The mixing efficiency is calculated at periodic intervals by processing the 10 seconds snapshots of the system. For the sake of consistent measurements, the multi-tracking peers that associate with *k* swarms are calculated as *k* distinct peers registering to one single swarm each.

5.2. Default Experiment Parameters

All the experiments are performed with steady state systems as mentioned earlier. The experiment is initialized so that initial peers in the swarm start with some share of the content hence emulating a steady system just after bootstrapping.

The peers are assumed to stay in the system until receiving a complete copy of the content. Upon download completion the peers may decide to stay in the torrent or leave instantly depending on the seeding time parameter. Only one publisher exists in the torrent and registers to all of the trackers. The publisher stays connected to the system during the whole experiment so all of the swarms are guaranteed to avoid starvation due to missing pieces. Peers have limited upload bandwidth but their download bandwidth is unlimited. The peers join the system according to a Poisson process or to an arrival-per-departure pattern.

The following default parameters are used for the experiments unless otherwise specified. Peers share a single file of 50MB that is split into 200 pieces with 256kB size each. The peers are allowed to upload with maximum 80KB/s rate. Although download rates are unlimited, the peers will be able to download at approximately 80KB/s when only leechers exist in the swarm as the available bandwidth is bounded by upload capacities. Peers are allowed to upload to 6 peers simultaneously but can receive from as many peers as they can. The upload rate limit of publisher can vary between experiments to provide seed power fairly to each experiment. For instance if the publisher is allowed to upload with 30KB/s to 6 peers in an experiment with 30 peers, it can upload by 60KB/s to 12 peers for the experiment with 60 peers therefore the effect of publisher is similar for each experiment.

The peer arrivals are generated with the arrival-per-departure method in most of the experiments as it decreases the effects of randomization while generating exponentially distributed inter-arrival times. The peers are set to leave the system as soon as they receive the file. Use of selfish peers sets the seed to leecher ratio to 0 hence keeping things simpler.

The peers are allowed to connect up to 50 peers by default. However connection list size can be adjusted according to the size of the experiments. The peers announce to the trackers every 1 minute. The number of peers requested from the trackers is set to 20 in order to increase the weight of peer exchange in peer discovery. The peer exchanges are performed every 30 seconds, 8 candidates for exchange are picked. A single PEX message is allowed to transfer at most 200 contact addresses; however size of messages is also bounded by the size of connection list as addresses are selected from connection list. The direction of the peer exchange is set to random, however the migrating peers can be forced to prefer candidates from own swarm so that the information from previous swarm can be injected to the newly arrived swarm.

The peers log their statistics every 10 seconds which provides high accuracy in measurement while not generating excessively large log files. The network emulator, ModelNet, is disabled for all the experiments.

The experiments are performed by varying number of swarms, size of the torrent and migration/multi-tracking willingness of the peers. The load per computer is kept equal for the experiments whenever possible, especially for throughput measurements. The number of slaves used increases as the torrent size increases while running the same number of peers in each machine.

The experiments with Poisson process are repeated 10 times to avoid misinterpretation due to stochastic behaviour or longer runs are preferred. On the other hand when arrival-per-departure pattern is used 3-4 repetitions are considered sufficient for accurate measurement.

Table 4 represents the most frequently used experiment parameters so readers can get familiar with the setups and parameter abbreviations.

Table 4 : ***Default Experiment Parameters***

content size	=	50 *MB*
piece size	=	2^18 *bytes*
torrent size	=	*vary from* 15 *to* 300 *peers*
number of swarms	=	1 to 8 *swarms*
number of publishers	=	1 *(registering to all swarms)*
publisher capacity	=	*adjusted with torrent size to have fair capacity*
migration/multi-tracking willingness (β)	=	*vary from* 1 *to* 8

arrival pattern = arrival-per-departure (some experiments use Poisson arrivals)

mean inter-arrival-time = estimated by Little's law or arrival per departure for each different setup if Poisson arrivals are used

mean seeding time = 0 *(peers leave upon completion hence Seed/Leecher ratio =* 0*)*

number of slaves = torrent size / load per machine (if equal load experiments are aimed)

number of runs = 5 *(can be increased if Poisson arrivals is used)*

upload rate = 40-80 *KB/s*

download rate = ∞ *(unlimited)*

max uploads = 6 *(maximum number of simultaneous uploads)*

connection list size = 50 *peers (can vary between 20 and 50)*

tracker announce interval = 1 *minute*

tracker numwant = 20 *peers (can vary between 20 and 50)*

peer exchange interval = 30 *seconds*

max pex candidates = 8

max pex message size = 200

direction of peer exchange = random (migrating peers can be forced to prefer own swarm)

peer log statistics interval = 10 *seconds*

ModelNet = disabled

Table 4: Default Experiment Parameters

The swarm management algorithms are represented as RMT (p, β, k) and RPM (p, β) where p is the size of the connection list, β is the willingness parameter and k is the number of trackers that a multi-tracking peer associates with. The k parameter can be set to n which means the peer connects to all available trackers.

5.3. Results

The swarm management algorithms are evaluated in scenarios where existing swarms are self-sufficient so that none of them heavily suffers from piece unavailability and consequently from peer contribution. Considering torrents in which swarms have different properties such as one swarm starving from piece unavailability due to publisher inexistence while the other swarm possesses all the pieces, mixing algorithms will help the swarms to avoid starvation. Furthermore considering swarms with different sizes where small swarms suffer from peer participation, discovering peers from other swarm would certainly improve performance of the suffering swarm.

It is clear that mixing algorithms help increasing overall performance if at least one swarm succeeds to perform well while some others are suffering by virtually integrating the suffering peers to the well performing swarm. The swarm management causes an increase in protocol overhead however it can be tolerated when the gain in stumbling swarms are considered. The decreasing download time amortises the cost of swarm management in some of the cases. As there is no need to restate the obvious, the experiments in which performance increase is easily predictable are not performed. Instead the algorithm performances are evaluated in torrents with equivalent swarms where each swarm can survive by itself.

The experiments focus on two metrics: mixing efficiency and download times while varying the size of the torrent or number of trackers. It is easy to observe the mixing efficiency; however to be able to observe download performance two important parameters, number of peers running per machine and average node degree of peers should be kept stable. Both parameters are important as they are directly related with the load on computers hence affecting the overall performance of the system. Running more peers on a machine decreases the performance due to increasing load. On the other hand, a peer interacting with more peers consumes more resources therefore increases the load on computers. The effect of average node degree gets more visible as more peers are running on computers. In general the BitTorrent protocol is known to perform better with increasing node degree [24]; where maximum performance can be observed with a clique like overlay, in other words when all peers know each other. However under heavy load the opposite case can be observed due to insufficient resources.

Running equal number of peers per machine can easily be handled by adding new slaves for increasing torrent sizes with equal steps. However keeping the average node degree can be troublesome in some cases. The average node degree of peers is strongly correlated with two parameters, the maximum size of connection list and speed of peer discovery. In order to keep the average node degree similar while changing the swarm size, the speed of peer discovery should be determined accordingly. A possible solution is to limit the size of connection list to the size of smallest swarm and adjust the number of peers requested from tracker and frequency of tracker queries. For instance one can observe torrent sizes of 20, 40, 60 etc. by limiting the connection list to 20. With mentioned settings, the experiments will end up with an average node degree close to 20 as connection lists of all peers will be kept fully utilized during the experiment by tracker queries or peer exchanges.

Unfortunately if the connection list is limited with the size of smallest swarm, the observable range of mixing efficiency narrows down. Considering a torrent with 3 equally sized swarms where peers are allowed to connect up to a single swarm of peers, the maximum value of mixing efficiency is around 0.67 according to the definition of mixing efficiency, the case when peers only connect to external peers which is highly unrealistic.

The limited number of available computers obviously constrains the scale of experiments. The sensitivity of experiments to average node degree decreases if the number of peers running on each machine decreases. If computers are not fully loaded, it is easier to observe performance improvement as the performance loss due to load increase can be neglected.

Another solution applied to overcome problem of load imbalance between experiments is to include all available computers to the experiment and have a constant torrent size. Then one can adjust the size of swarms by changing the number of available trackers. As peers with vanilla behaviour pick swarms uniformly at random, swarms will be equally sized on average. Then the mean values of per swarm statistics can be collected for various swarm sizes while keeping the load balanced for experiments.

Therefore observing the mixing efficiency and download times as a function of torrent size can be considered challenging. Our experiments were performed such that the two metrics could be observed. Some of the experiments only focus on the mixing efficiency as showing the increase in swarm sizes one can assume the performance improvement.

It should be mentioned that the willingness parameters of the initial peers are always set to 0 as they join the system in a very short interval. If initial peers were assigned a willingness value, more peers than expected would exhibit mixing behaviour as the peers will observe smaller values for torrent size during the bootstrap process.

5.3.1. Protocol Performance as a Function of the Torrent Size

These experiments aim to show that BitTorrent performance improves with increasing torrent size. *Figure 16* represents the average download time of the peers in a torrent as a function of torrent size.

The data points are collected through running experiments with torrent sizes 30, 45, 60 and 75 and a single tracker is used. The connection list size is set to 50. The loads of experiments are kept equivalent by increasing the number of slave machines for larger swarms while running 15 peers per machine. The results show that larger torrents perform better than smaller ones.

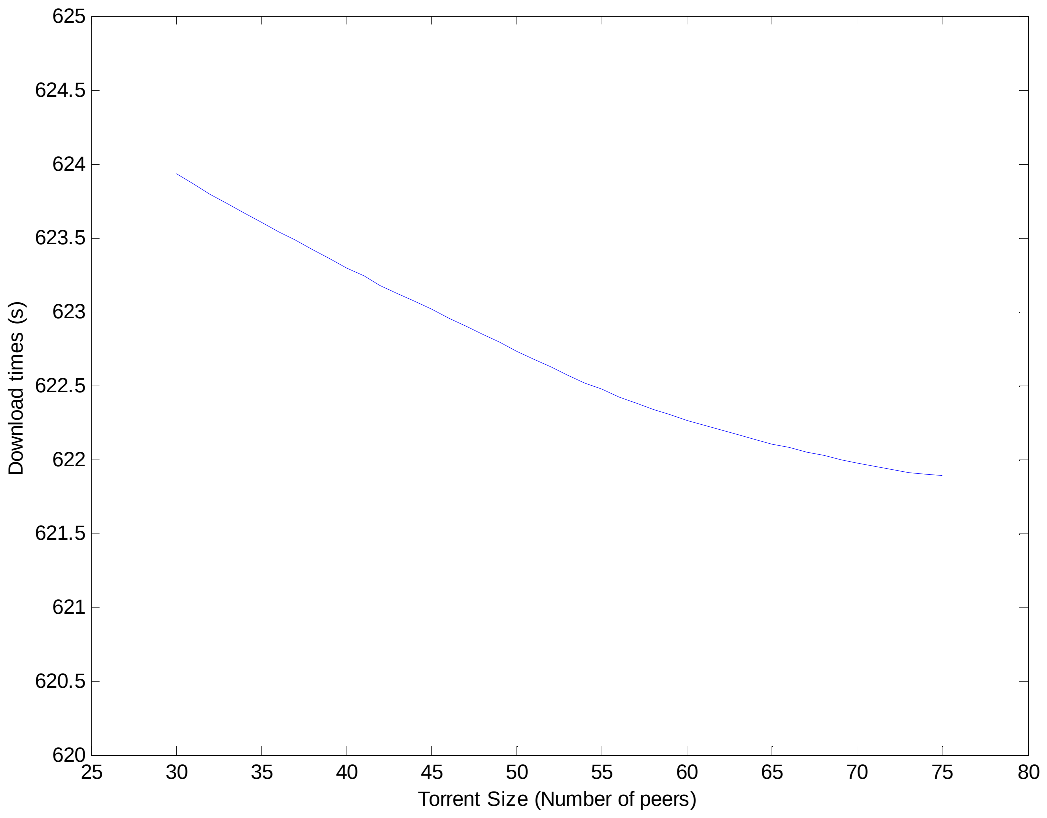


Figure 16: Protocol Performance vs. Torrent Size with Arrival-per-Departure Pattern

The experiment is also performed with peer arrivals according to Poisson process. The upload capacity of peers is decreased to 40KB/s for this experiment to have longer runs so that effects of stochastic arrivals can be avoided. The swarm sizes are set as 60, 90, 120 and 150. The mean inter-arrival times for Poisson process are calculated by running the same setup with arrival-per-departure arrivals and they are calculated as 22.3, 14.95, 11.15 and 8.9 seconds respectively. The peers are allowed to connect to at most 50 peers. *Figure 17* shows the average download time of peers as a function of swarm size when Poisson arrivals are used.

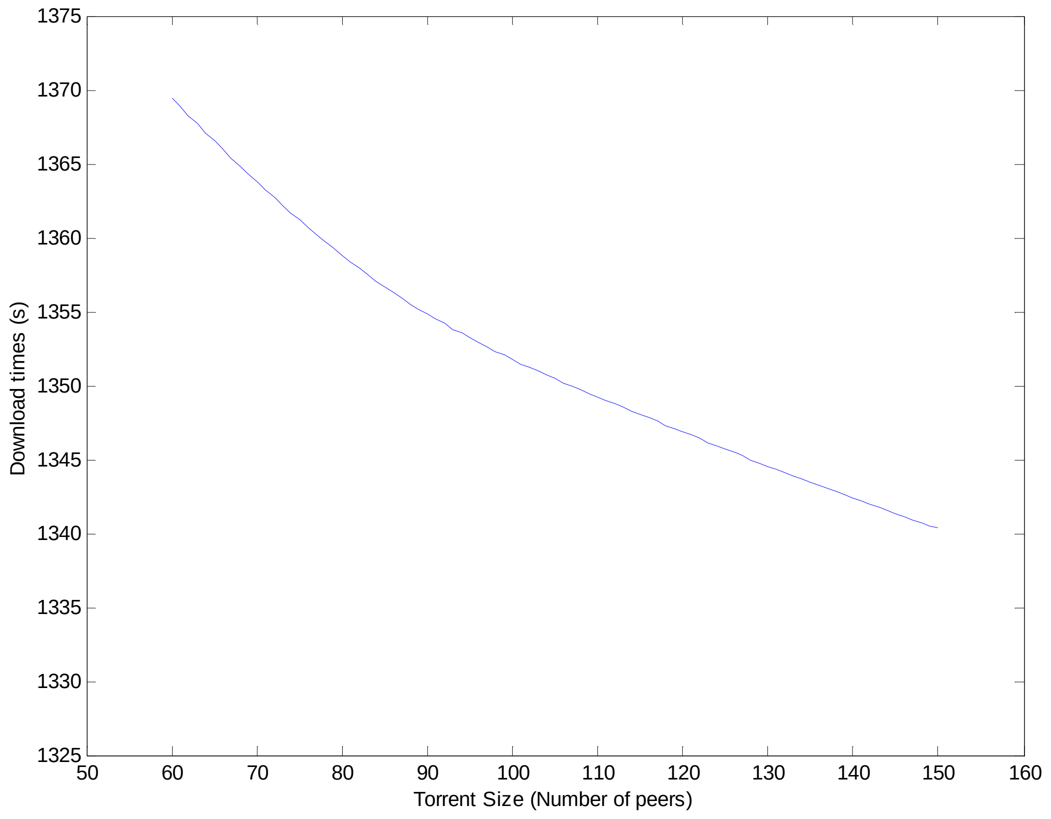


Figure 17: Protocol Performance vs. Torrent Size with Poisson Process Arrivals

The protocol performance as a function of swarm sizes can also be evaluated by varying the number of trackers. For a torrent of 250 peers, setting the number of trackers to 1, 2, 4, 6 and 8 the performance of swarms with an average size of 250, 125, 62.5, 41.67 and 31.25 can be evaluated. With this setup, all experiments are guaranteed to be performed under the same load while the swarm sizes are varied as the same set of machines (preferably all available computers for lowest load) are used for each experiment. Figure 18 shows the average download times of peers in a torrent with 250 peers as a function of swarm size which is varied by adjusting the number of trackers.

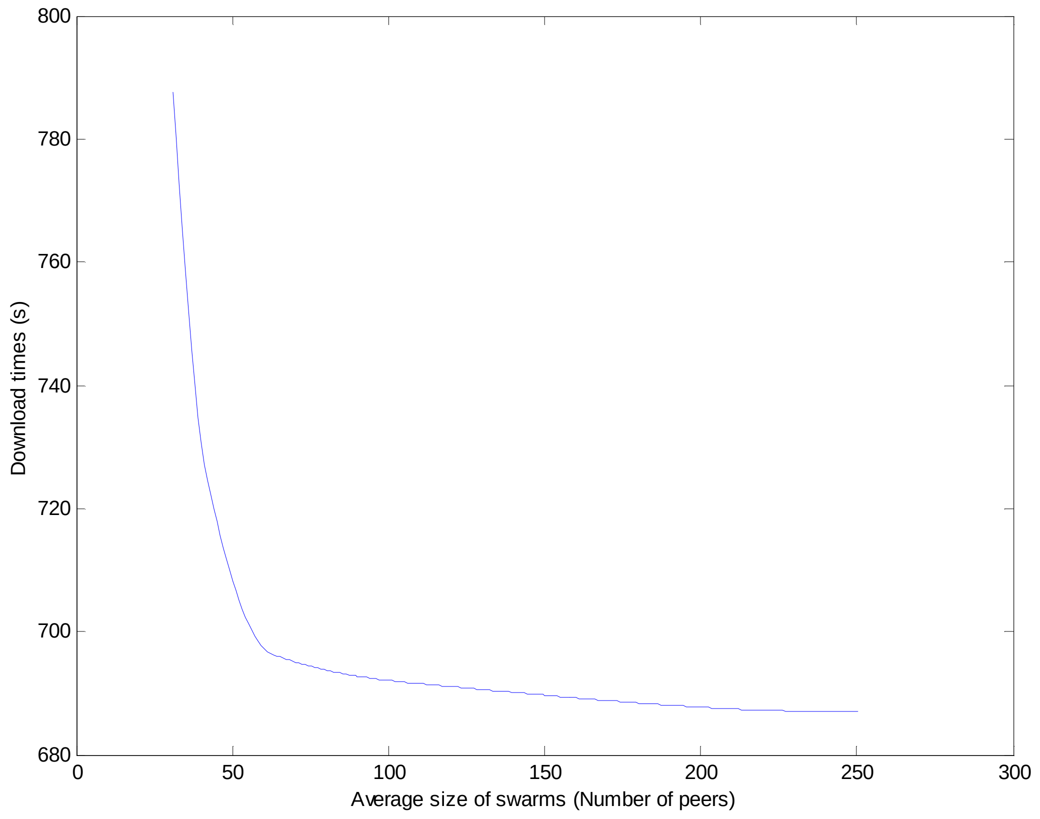


Figure 18: Protocol Performance vs. Swarm size as a function of number of swarms

Figure 18 again shows a performance improvement with the swarm size. As can be seen the protocol performs best when all peers are in a single swarm which can be achieved by e.g. using the pick biggest swarm algorithm. The significant performance degradation in swarms having approximately 30 peers can be explained by the low piece availability in the swarm. Due to small number of peers, it is possible that some pieces are only possessed by the publisher. In such situations, the available pieces in the swarm are easily exchanged among peers while missing pieces should be sent to the swarm by the publisher hence limiting the overall system performance by the publisher's capacity.

Another important result that can be of interest is that the download times of peers are normally distributed which implies that homogeneous peers are fairly served in BitTorrent. As an example, the distribution of peer download times of experiments with Poisson process is given in *Figure 19*. The download time distributions are tested with Arena Input Analyzer tool and they are suggested to best fit to normal distribution. The results successfully passed the Kolmogorov-Smirnov [25] test at significance level 0.05 performed by the tool. The figure represents the download time distribution of two sample runs with 60 peers and 150 peers as a QQ plot versus standard normal.

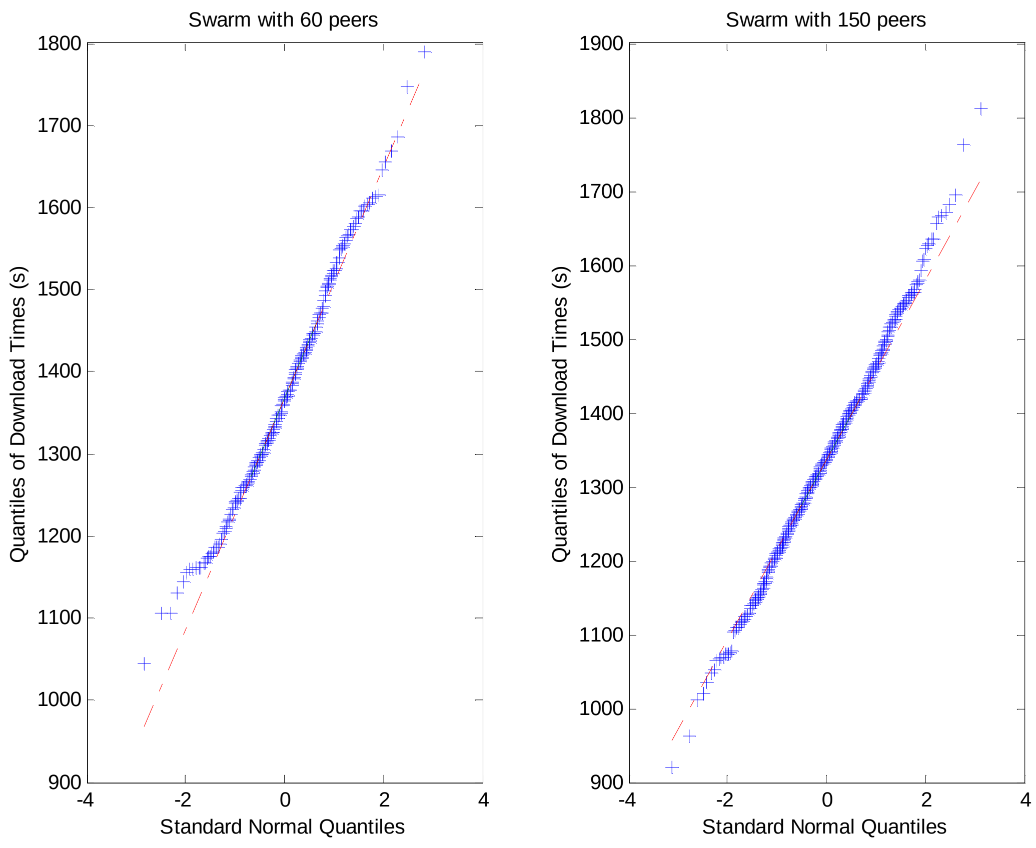


Figure 19: QQ Plots of Download Times vs. Standard Normal

The experiments in this section were performed in order to verify the correctness of the testbed. The results indicate that increasing the swarm sizes improves the performance. This observation justifies the use of swarm management algorithms, because swarm management algorithms aim at virtually increasing the size of swarms via mixing peers in different swarms.

5.3.2. Experiments with 2 Swarms

The swarm management algorithms, RPM and RMT, are evaluated in a torrent with 2 swarms. The torrent size is varied as 15, 30, 60, 120, 180, 240 and 300 and peers are allowed to connect at most 50 peers. The tracker queries are performed every 1 minute and 20 peers are requested from the tracker. The tracker numwant parameter is decreased to 20 in order to increase the weight of PEX protocol in peer discovery so that mixing of swarms can be observed better.

This experiment only focuses on mixing efficiency. The download performances of peers are ignored as the experiments could not be run under equal load. The torrents with at least 60 peers run 60 peers per computer however smaller ones have to be examined under lower load. The peers have their upload bandwidths limited at 80KB/s. The willingness parameter (β) is set to 1, 2, 4 and 8 and we measure the mixing performance of the algorithms.

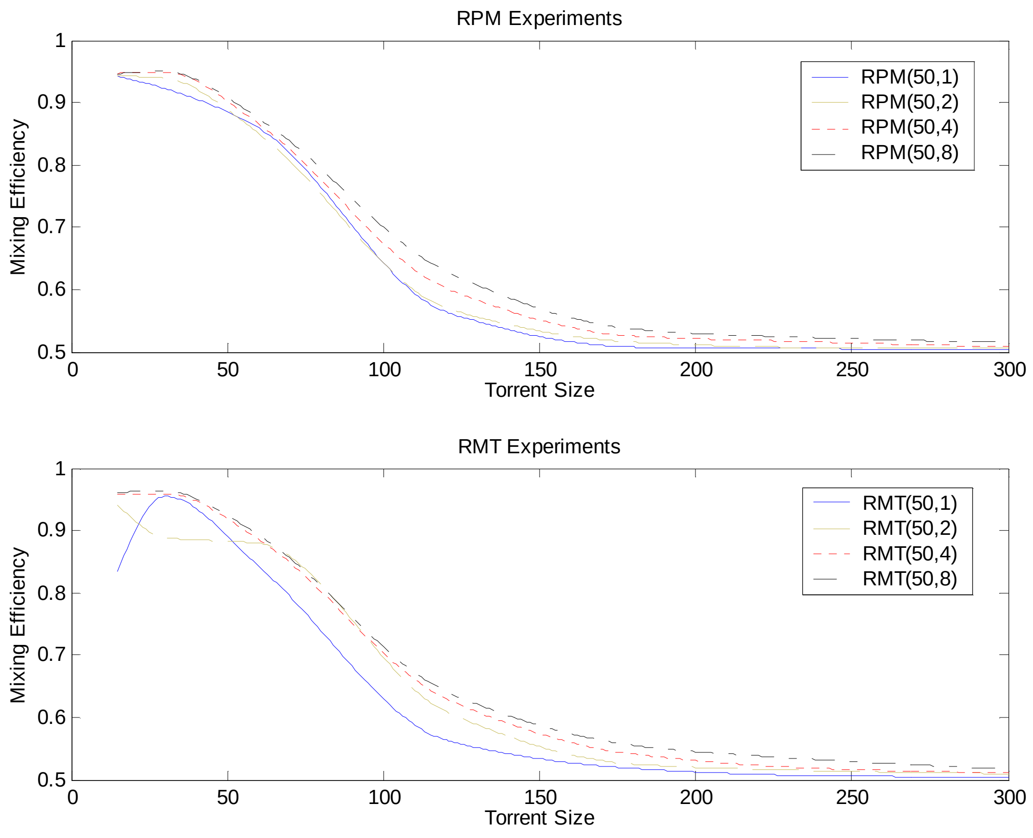


Figure 20: Average mixing efficiency as a function of torrent size for RPM and RMT with changing willingness

Figure 20 shows that both algorithms can achieve a very high mixing efficiency, around 0.95 for small torrents meaning that peers nearly discover all other peers in different swarms. The mixing efficiency decreases with increasing torrent size, because the size of the connection list limits the number of peers that can be discovered. For torrents with 300 peers the mixing efficiency is close to 0.51 which is almost negligible as without mixing, a torrent with two equally sized swarms will have the average mixing efficiency calculated as 0.5.

Increasing the willingness parameter obviously increases the efficiency of the algorithms but with a decreasing gain. Therefore the willingness parameter should not be set very high as the gain may not be worth the linearly increasing load.

The deviations in RMT experiments with small torrents are simply due to the stochastic behaviour. In experiments with 15 or 30 peers it is probable that multi-tracking peers arrive much later than expected thus decreasing the mixing efficiency on the average. But with longer experiments these artefacts could easily be avoided as effects of stochastic behaviour will be decreased.

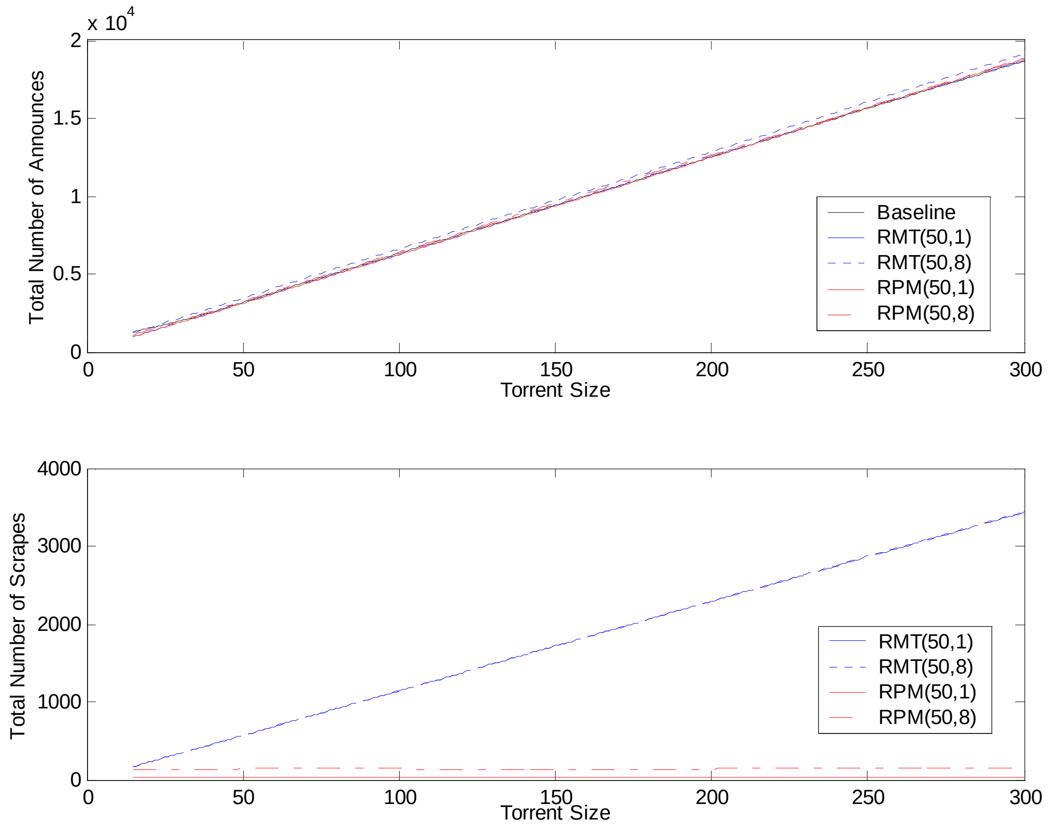


Figure 21: Tracker Load vs. Torrent Size

Figure 21 represents the total number of announces and scrapes as a function of torrent size. The total number of announces increases linearly with the number of peers intuitively. Although it is hard to observe in the figure due to large scale, careful eyes should notice that baseline algorithms generates more overhead for small torrents. The reason for this is simply the longer download times in small swarms without mixing in which the overhead due to mixing algorithms is amortised by the performance improvement. The load of algorithms gets higher with the increasing willingness. Although the difference is very small, it should be noted that the load of RMT is more than that of RPM. The scraping overhead of RMT increases with the torrent size as each peer scrapes all trackers upon arrival. However RPM's scraping overhead remains constant as the number of migrating peers does not change with torrent size. The number of scrapes for baseline algorithm is ignored as it is zero.

5.3.3. Experiments with 3 Swarms

The algorithms were also evaluated in torrents with 3 swarms. The experiments are performed as an extension of 3 swarm experiments and focus on the mixing performance of the algorithms. The experiments are performed with torrent sizes as 30, 60, 90, 150, 210 and 300 and the connection list size is set to 50, same as in the previous experiment. The tracker query interval is 1 minute and 20 peers are requested just like in the 2 swarm experiments. The willingness parameter (β) is set to 1 and 8. The number of trackers that a multi-tracking peer registers (*k*) is set to 3.

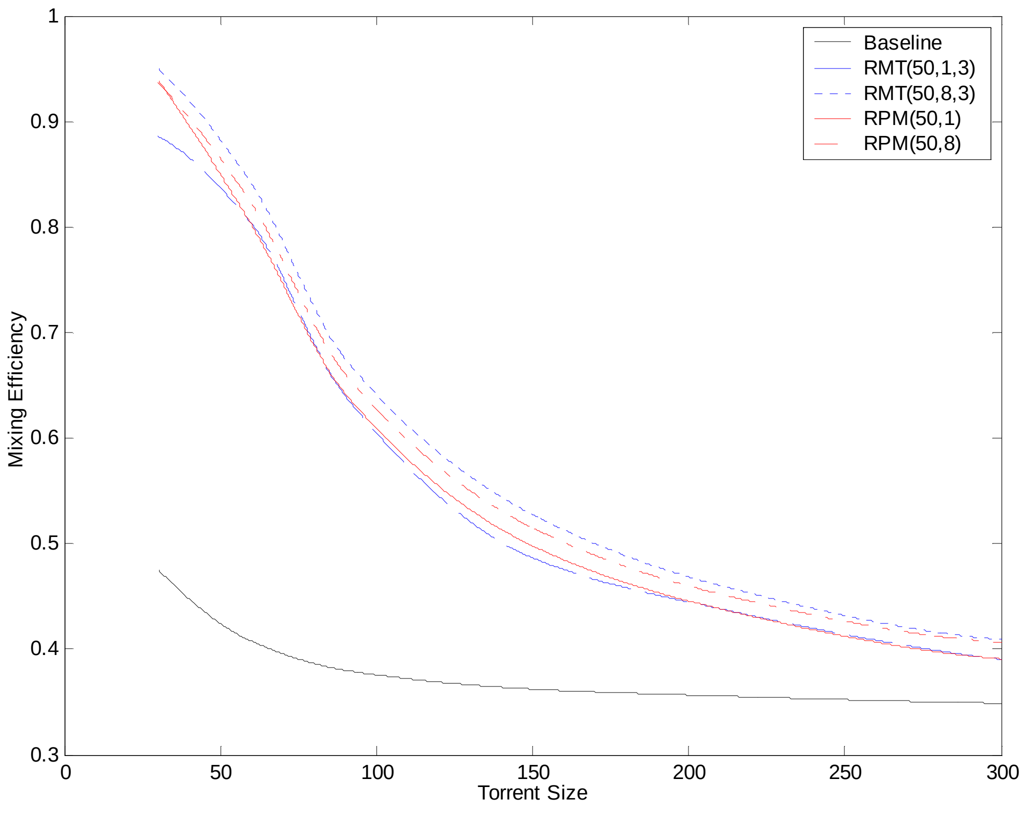


Figure 22: Mixing efficiency vs. Torrent Size

Figure 22 shows the mixing performances of algorithms. The results of baseline experiment are added to show the lower bounds as no mixing is performed with baseline algorithm. The mixing decreases with the increasing torrent size due to limited connection list size. As mentioned earlier, the mixing improves with the willingness parameter but with a decreasing marginal gain so setting it too high should be avoided.

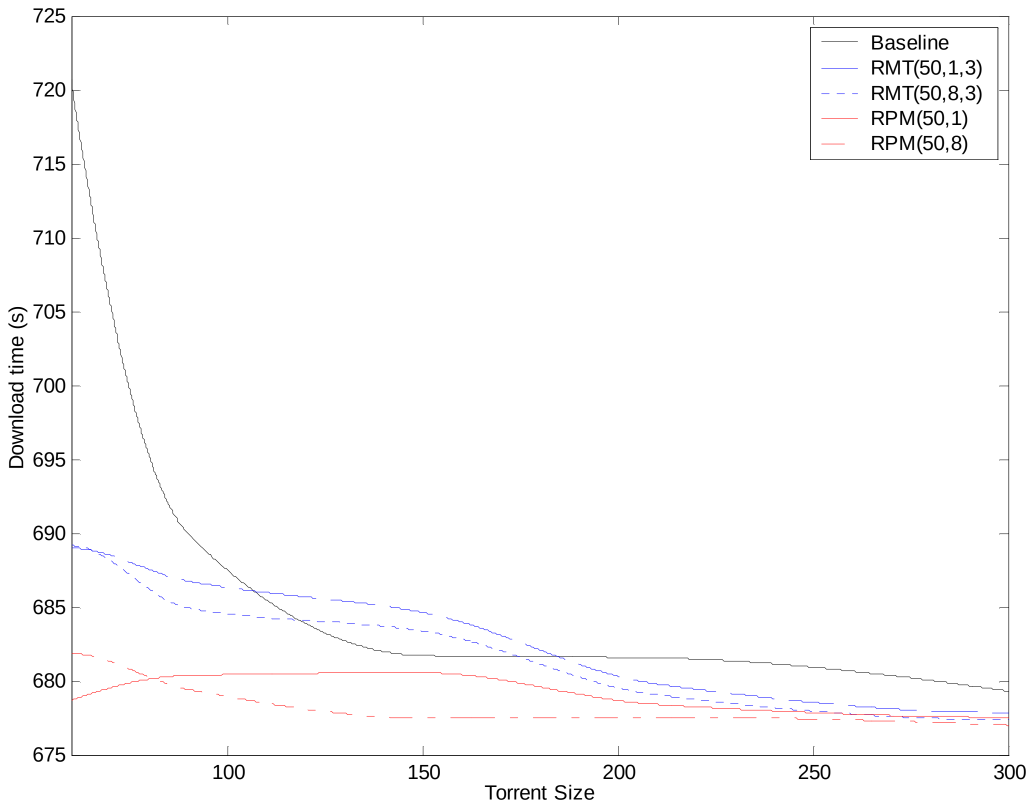


Figure 23: Download performance of algorithms as a function of torrent size

Although the experiments are not performed under the same exact load, *Figure 23* shows the download performance of algorithms where the load on computers can be considered as close. In most of the cases, the mixing algorithms help improving download performance but the effect is much more visible for smaller swarms. As can be seen, the baseline algorithm shows poor performance for small torrents.

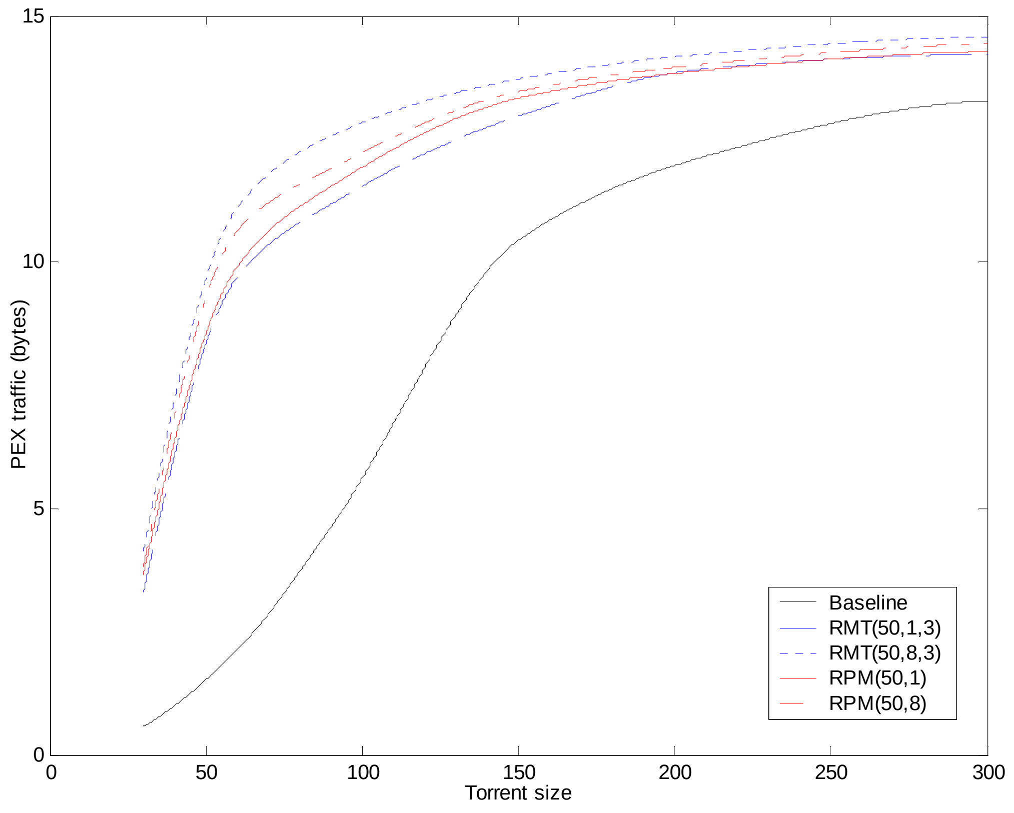


Figure 24: PEX traffic per peer vs. Torrent Size

Figure 24 represents the average peer exchange traffic generated by a single peer. The PEX traffic is higher when mixing algorithms are used as they utilize the PEX messages, more peers are appended in messages. The PEX traffic increases with the increasing swarm size but it converges due to the connection size limitations.

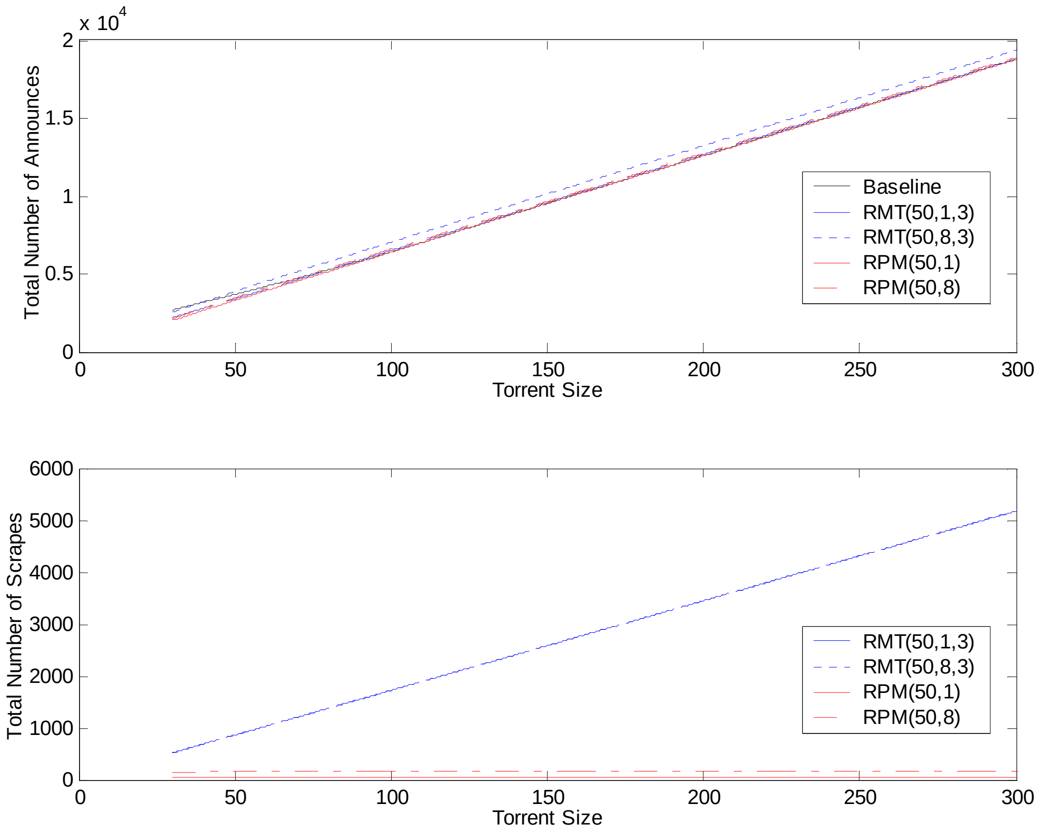


Figure 25: Tracker load vs. Torrent Size

Figure 25 represents the load on trackers as a function of the torrent size. The results are very similar to the experiments with 2 swarms. The load when using the baseline algorithm is higher than when using mixing algorithms for small torrents due to high download times. The tracker load on a swarm with 3 torrents is slightly higher than the 2 swarm torrent. The effect of swarm size on tracker load will also be examined in following sections.

5.3.4. Performance as a Function of the Number of Swarms

The performance of swarm management algorithms is evaluated against the number of available trackers. Given a fixed torrent size, performance of baseline behaviour, RPM and RMT algorithms with different willingness are observed as a function of number of swarms.

Firstly the algorithms are evaluated with a torrent of 100 peers while setting the number of swarms to 1, 2, 4, 6 and 8. All available computers are used for the experiments thus the average number of peers running on each machine is set to 20. As the torrent size remains stable the experiments are performed under same amount of load. The connection list size is set to 50 and tracker replies contain 20 peers to increase the weight of peer exchange in peer discovery so that mixing can be observed better.

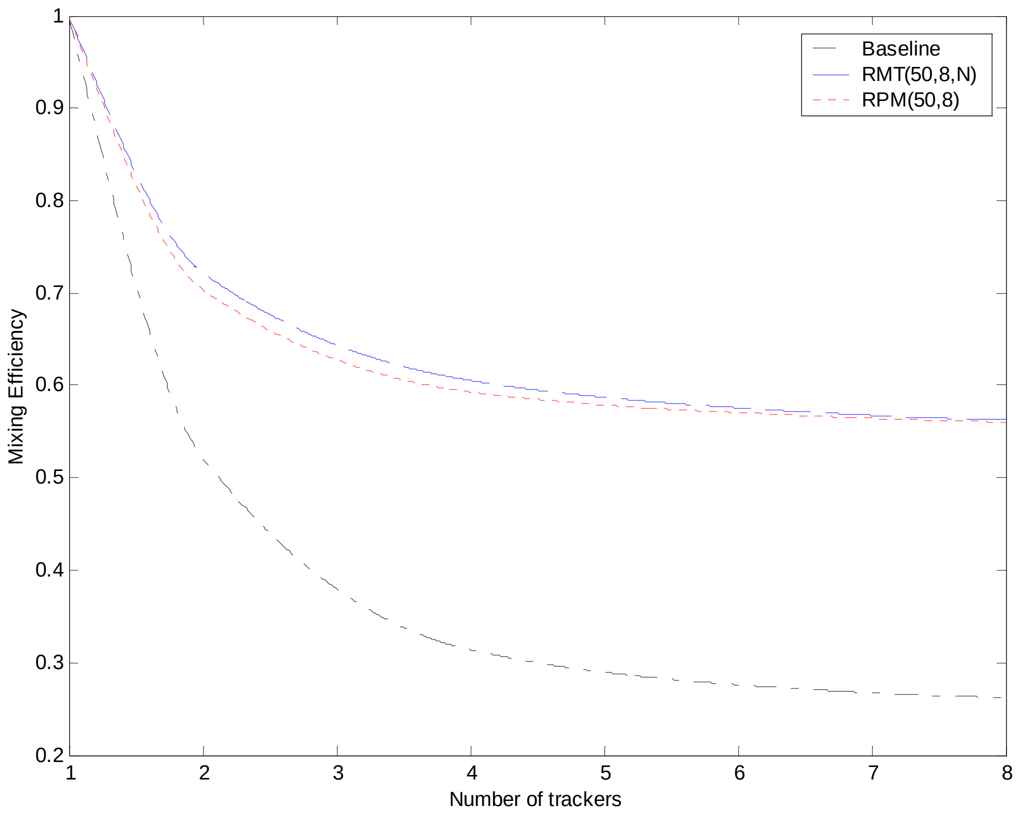


Figure 26: Mixing Efficiency vs. Number of Trackers for 100 peers torrent

Figure *26* represents the mixing efficiency of the algorithms as a function of the number of swarms. The baseline experiment is added to show the lower bounds of mixing efficiency when no mixing is performed. The mixing efficiency is set to 1 for a single swarm and it decreases with increasing number of swarms.

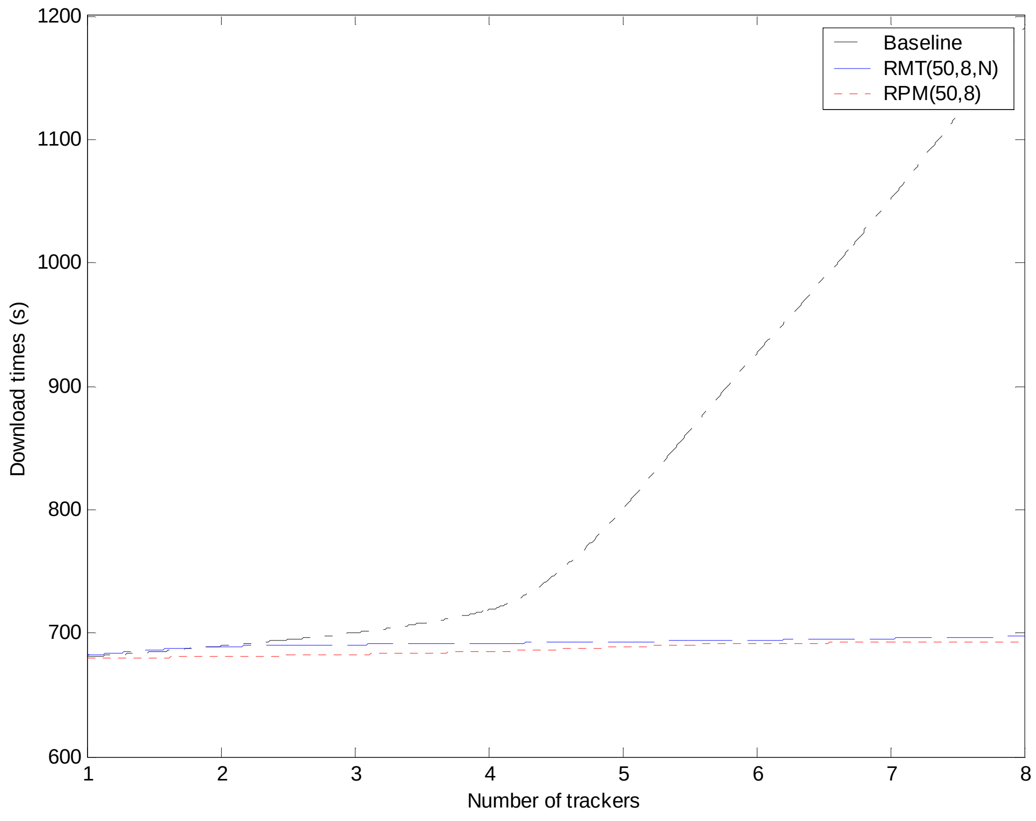


Figure 27: Average download times of peers vs. Number of trackers

In *Figure 27* the average download times of peers are shown for increasing number of swarms. The baseline algorithm starts performing poorly as the size of the swarms gets smaller (increasing number of trackers). The dramatic increase in download time for 6 and 8 trackers shows that the protocol starts suffering from piece unavailability and peer contribution when average size of swarms drops under 25 peers. The algorithms perform very similar when peers are in a single swarm as expected. Use of mixing algorithms successfully prevents performance loss even when the average size of swarms gets smaller. However slight performance degradation is still observed with increasing number of swarms due to decreasing mixing efficiency.

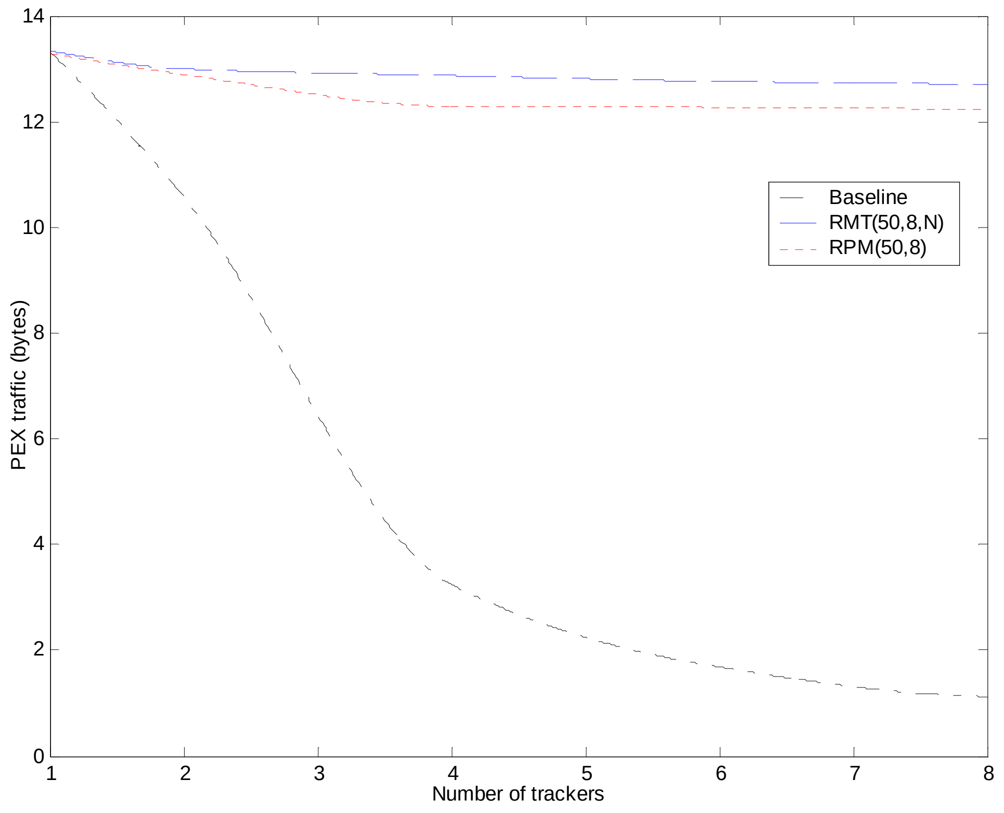


Figure 28: Average PEX traffic per peer vs. Number of trackers

Figure 28 shows the average PEX traffic generated per peer. The PEX traffic in baseline algorithm decreases when the swarms are smaller as the available contact addresses to exchange decreases. However RPM and RMT algorithms keep the PEX traffic at similar amounts as peers are introduced to each other via mixing. Therefore they can exchange peers as if they were in a single swarm. The small decrease in PEX traffic with increasing number of swarms is due to decreasing mixing efficiency as the size of virtual swarm decreases.

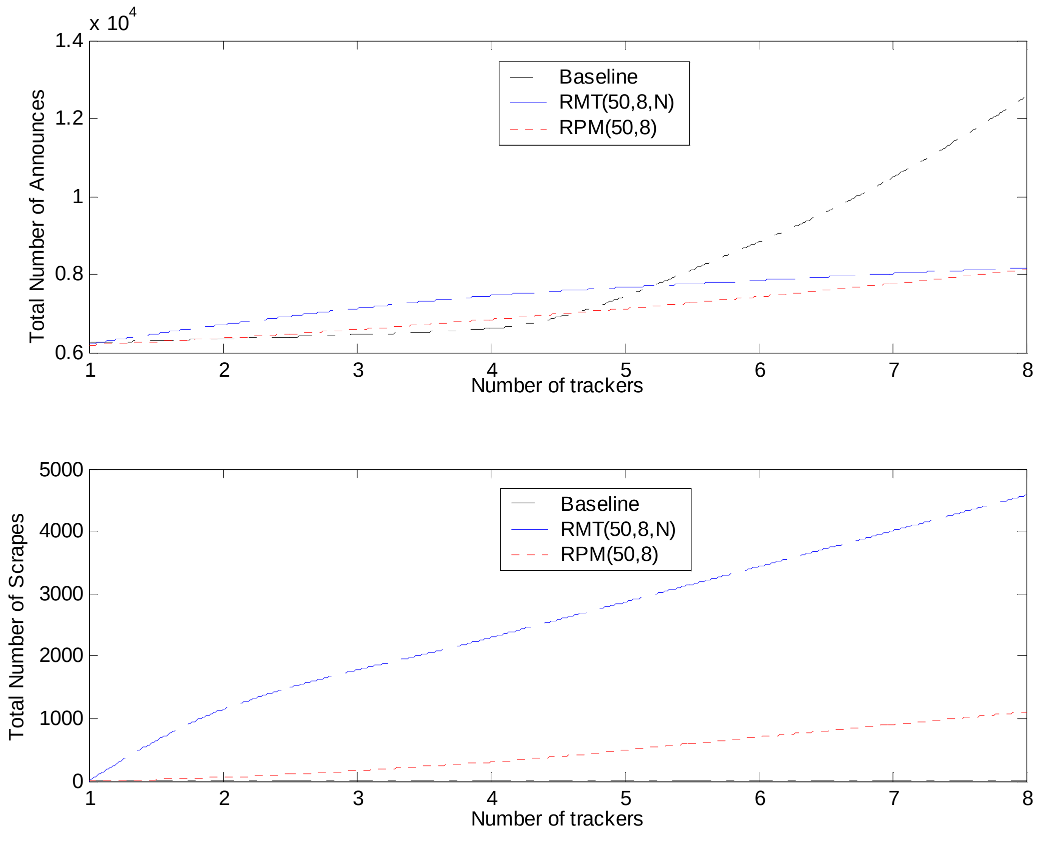


Figure 29: Aggregate Load on Trackers vs. Number of trackers

Figure 29 shows the total number of announces and scrapes performed by the peers regarding the number of swarms. The baseline algorithm causes lower overhead when it performs close to the mixing algorithms. However when it starts to perform poorly, the overhead caused by baseline peers can be higher than others as the total time spent in the system increases which causes an increase in the number of queries sent by each peer. The baseline algorithm does not cause any scraping traffic. The traffic caused by RPM is directly related with the number of migrations as each migration performs a scrape and unregister/register messages. As known the frequency of migration checks increases with number of available swarms since the portion size is defined as $1/(\beta(|R|-1))$ of the total content. In addition to that the average number of peers migrating from a particular swarm is independent of its size. Therefore increasing number of swarms causes more peer to migrate consequently increasing the load on trackers in terms of both announces and scrapes. As mentioned earlier the load increase caused by RMT algorithm is proportional to $n\beta$ as in steady state the system is expected to have $n\beta/k$ peers connected to k trackers. In the figure the tracker announces of RMT decreasingly increases with the number of trackers. The decrease in increase is caused by the difference in torrent sizes between the experiments. Although the experiments are known to be performed with 100 peers, every multi-tracking peer increases the torrent size with k (equal to n for this experiment). The increasing torrent size lowers the probability of new multi-tracking peer arrivals as they are indirectly proportional. Therefore with increasing n, the number of peers that multi-track decreases while the load caused by them increases. The scraping overhead of RMT is rather straightforward. Upon arrival, each peer scrapes all available trackers hence the scrape load linearly increases with number of trackers.

The experiment above is also repeated with a torrent of 250 peers by increasing the load per machine to 50 peers. Different from the experiment above this experiment focuses on determining a suitable k

value for random multi tracking. The number of trackers (hence, the average size of swarms) is varied for algorithms RMT(50,1,2), RMT(50,8,2), RMT(50,1,*n*) and RMT(50,8,*n*).

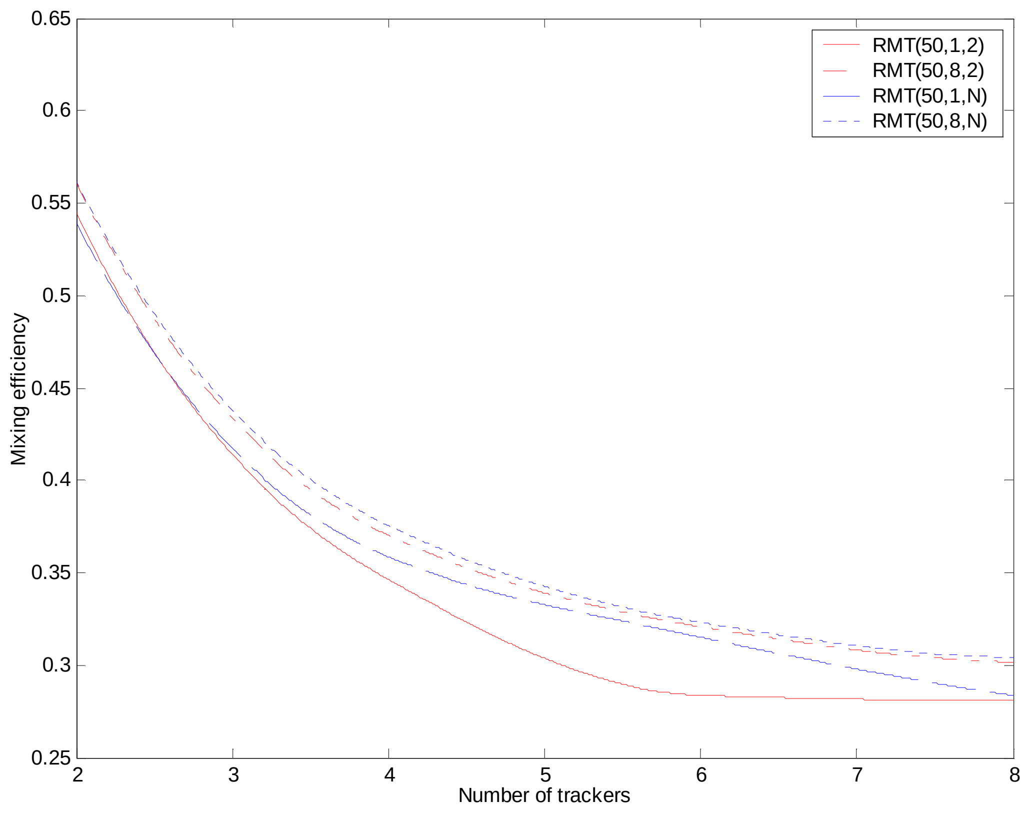


Figure 30: Effect of k on mixing efficiency

Figure 30 shows the effect of *k* on the mixing efficiency. RMT performs very close even for two ends of *k* value, 2 and *n*. However in most of the cases maximizing *k* seems to provide better mixing. The only situation where setting *k* to 2 performs better is runs with 2 swarms in which actually both algorithms are same. Therefore according to the results of experiments, maximizing *k* for RMT can be suggested. Considering the developed analytical model, one can conclude that PEX is efficient enough so that increasing *k* does not have effect on the efficiency of mixing. With this assumption, the model also suggests maximization of *k*. Another benefit of maximizing *k* is ensuring the connectivity among all swarms even when the number of multi-tracking peers is very low. Setting *k* to 2 may fail to connect all swarms if multi-tracking peers are unluckily assigned to redundant pairs of swarms when β is low.

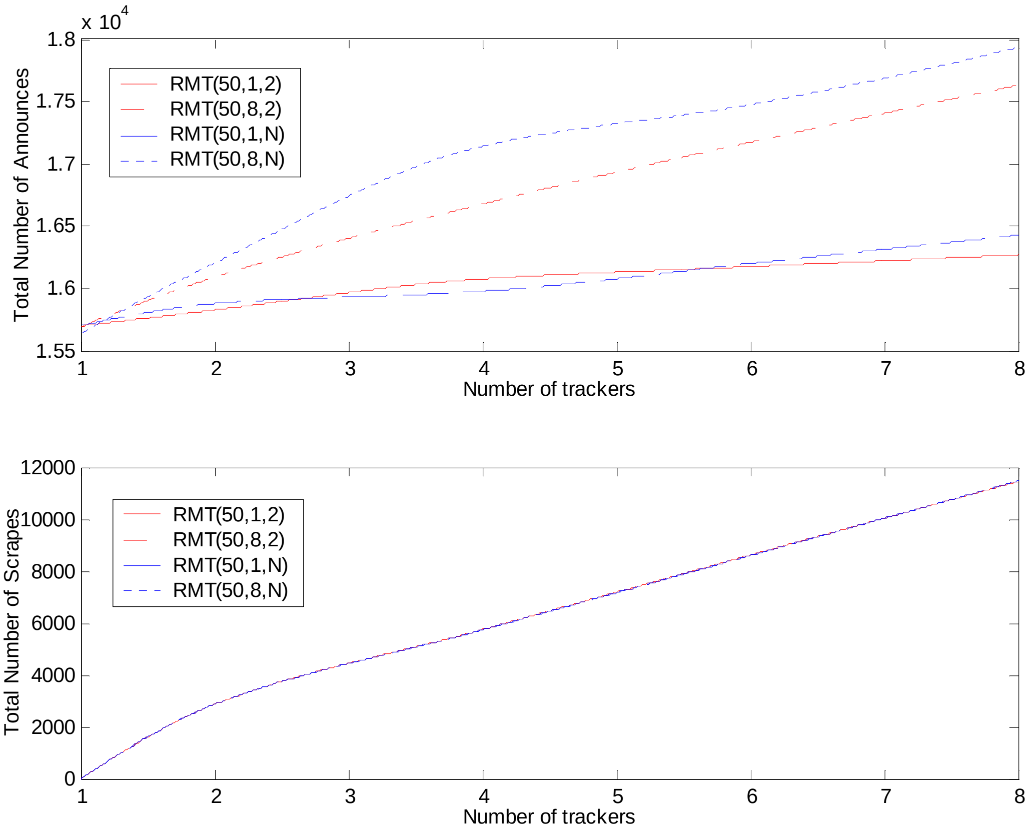


Figure 31: Effect of k on tracker load

Figure 31 shows the effect of *k* on the number of announces and scrapes. As mentioned earlier the load on trackers caused by RMT is proportional to *nβ* and independent of *k*. For β=1, the results strongly hold with the model as there is not an evident difference between algorithms. For β=8, setting *k* to *n* seems to cause more overhead then setting *k* to 2 which is an unexpected result. This result can be explained by the deviations in total number of multi-tracking peers in the torrent where even a single multi-tracking peer can increase the total number of announces by approximately 10 * *n*. The total number of multi-tracking peers will also be presented below. The total number of scrapes is almost same for each algorithm as it is actually determined by the number of peers in torrent which is very similar for the experiments.

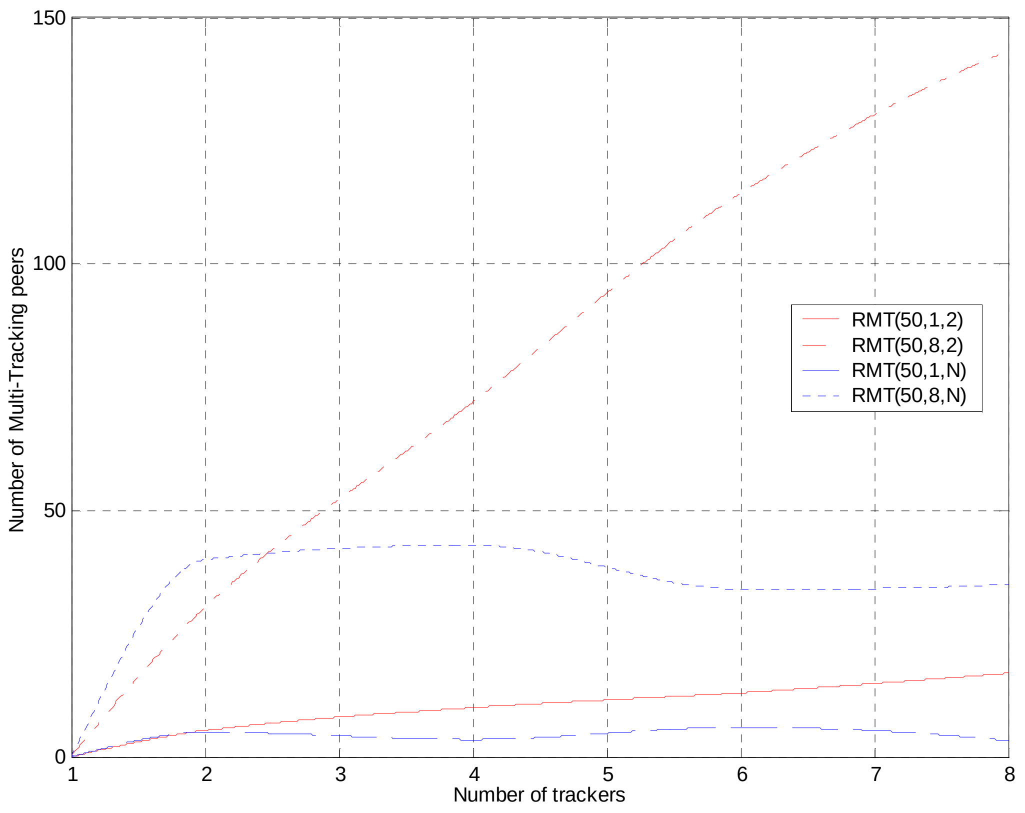


Figure 32: Effect of k on number of multi-tracking peers

The RMT protocol by its nature adjusts the number of multi-tracking peers accordingly so that the protocol overhead is independent of *k*. According to the protocol definition, the number of multi-tracking peers when *k* is set to 2 should be $n/2$ times the number of multi-trackers when *k* is *n*. However due to stochastic behaviour, the multi-trackers in *k*=*n* experiments is higher than the expected, or the number of multi-trackers for *k*=2 is lower, for some of the experiments which can be the reason for the load difference between experiments. Additionally when *k* is set to *n*, the number of multi-tracking peers is independent of number of swarms whereas the number of multi-tracking peers increases linearly with *n* for *k*=2. *Figure 32* shows the number of multi-tracking peers for the algorithms as a function of number of trackers. The number of multi-tracking peers that associate with *n* swarms remains constant whereas the peers that associate with 2 trackers increase linearly with *n*.

5.3.5. Discussion

The results presented above show that both random peer migration and random multi tracking successfully increase the connectivity between swarms, especially for small torrents. Increasing the sizes of swarms virtually, the algorithms increase the overall performance of the protocol. The gain in download times is estimated to be around 5% on the average for self-sufficient torrents. But when small torrents that starve from piece availability are considered the gain can be up to 40% as mixing can rescue these swarms from starvation.

Both algorithms perform very close in most cases however for increasing β it can be said that random multi-tracking (RMT) performs slightly better than random peer migration (RPM). The increase in protocol overhead as shown is kept at a modest level and even in some cases the protocol overhead due to mixing is amortised by the performance improvement. Even there is not much difference in

RMT’s performance for varying *k*; it is suggested to be maximized as the protocol gets less vulnerable to stochastic behaviour when β is low. Having a single peer that associates with all trackers can perform quite well whereas setting *k* to 2 may fail to connect all swarms when the number of multi-tracking peers is low. The β value should not be chosen too high as it increases the overhead linearly which does not worth the gain. The algorithms fairly distribute the overhead increase among trackers by their design hence mixing algorithms are shown not to cause an excessive load increase on a particular tracker.

The multi-tracking peers are noticed to perform slightly better when compared to vanilla peers. The reasoning behind this is simple. Registering to more swarms, multi-tracking peers become part of a larger swarm hence they always keep their connection list utilized. For *k*=*n*, the performance of multi-tracking peers increases by *n*, as registering to more swarms increases their chances to receive connection requests from other peers. As an expected result, the average PEX traffic generated by multi-tracking peers is above the average as they keep introducing peers in different swarms to each other by sending utilized PEX messages without redundancy. Interestingly, the download performance of migrating peers does not change deterministically but instead they behave like normal peers. The average PEX traffic generated by migrating peers is slightly below the average unexpectedly. The reason for this can be the imbalance between the number of peers known from each swarm. Upon migration, the list of migrating peer is mostly filled with peers from old swarm and only a few peers are known from the new swarm. As redundancy checks are performed, it is possible for the peer to send PEX messages with a few contacts to the peers in old swarm which decreases the overall average.

Even if the multi-tracking peers start performing slightly better than other peers, the download time distribution of peers still best fits the normal distribution. This implies that the mixing algorithms do not harm the fairness of the BitTorrent protocol for homogenous peers.

6. CONCLUSION and FUTURE WORK

This thesis presents the design and implementation of two dynamic swarm management algorithms Random Peer Migration and Random Multi Tracking, effective approaches to improve overall BitTorrent performance with low cost. The discussed swarm management algorithms increase the protocol performance close to the optimum (when all peers are registered in a single swarm), but without sacrificing the load balancing and resilience properties offered by multi tracker extension at the expense of only a small increase in communication overhead.

In order to measure the BitTorrent efficiency and gain of swarm management algorithms, a testbed was developed. The testbed allows performing controlled BitTorrent experiments by efficiently utilizing a set of computers. The experiments are aimed to be performed under realistic conditions as much as possible.

The algorithms are shown to improve download performance of the protocol around 5% on the average. However the gain can even be around 40% when starving small swarms exist. The algorithms are evaluated for various torrent sizes and number of trackers. The experiments show that mixing algorithms simply improve the protocol without harming other properties such as load balancing on trackers and robustness. Even the overhead increase is shown to be amortised by the gain in performance for small torrents.

Due to low number of available computers, the performance of algorithms could not be evaluated in larger scale experiments. As mentioned earlier; the load defined in terms of average node degree and number of peers per machine, should be balanced for each experiment which is not possible in some scenarios due to limited number of available computers. In order to keep the load balanced among experiments, the torrent sizes should be selected as multiples of load per machine.

With a slight modification the testbed can be deployed on a large scale platform such as PlanetLab or Emulab. With the available computers in those platforms, users can run large scale torrents by only running a single peer on each machine. In such a testbed, the algorithms can be evaluated in more detail as there would not be any resource constraints. Increasing the number of computers for the setup allows performing larger scale experiments while decreasing the load on the individual machines. Having more computers, the users can also experiment with a larger content which helps observing the protocol characteristics better.

The algorithms should also be evaluated in scenarios where peers exhibiting different behaviours coexist in the torrent. As peers perform the mixing behaviour with a probability correlated with the number of peers downloading the same content, the number of mixing peers would be less than the expected but the efficiency of them is still a question. After all the controlled experiments, the performance of the algorithms should also be assessed in the wild.

REFERENCES

[1] Cohen, Bram. "The Bittorrent protocol specification." 28 Feb 2008. Web. 10 Feb 2010. <http://www.bittorrent.org/beps/bep_0003.html>.

[2] "BitTorrent Still King of P2P Traffic." 18 Feb 2009. Web. 10 Feb 2010. <http://torrentfreak.com/bittorrent-still-king-of-p2p-traffic-090218/>.

[3] Yang, Xiangying, and Gustavo de Veciana. "Performance of Peer-to-Peer Networks: Service Capacity and Role of Resource Sharing Policies." In Proc. of IEEE INFOCOM, 2004

[4] "Bittorrent Protocol Specification v1.0." 7 Feb 2010. Web. 11 Feb 2010. <http://wiki.theory.org/index.php?title=BitTorrentSpecification&oldid=3392>.

[5] Cohen, Bram. "Incentives Build Robustness in BitTorrent.". *In Proc. of the 1st Workshop on Economics of Peer-to-Peer Systems*., 2003

[6] A.-L. Barabási. “Linked: The New Science of Networks”, Perseus Publishing, 2002.

[7] Neglia, Giovanni, Giuseppe Reina, Honggang Zhang, Don Towsley, and Arun Venkataramani. “*Availability in BitTorrent Systems*.” in Proc. of IEEE INFOCOM, 2007.

[8] Pouwelse, Johan, Pawel Garbacki, Dick Epema, and Henk Sips. "The Bittorrent P2P File-sharing System: Measurements and Analysis." in Proc. of IPTPS, 2005.

[9] "P2P:Protocol:Specifications:Multitracker." Web. 12 Feb 2010. <http://wiki.depthstrike.com/index.php?title=P2P:Protocol:Specifications:Multitracker&oldid=2304>.

[10] Maymounkov, Petar, and David Mazieres. "Kademlia: A Peer-to-peer Information System Based on the XOR Metric." in Proc. of IPTPS, 2002.

[11] Stutzbach, Daniel, and Reza Rejaie. "Improving Lookup Performance over a Widely-Deployed DHT." in Proc. of IEEE INFOCOM, 2006.

[12] "Kademlia: A Design Specification." Web. 16 Feb 2010. <http://xlattice.sourceforge.net/components/protocol/kademlia/specs.html>.

[13] Loewenstern, Andrew. "DHT Protocol." 28 Feb 2008. Web. 17 Feb 2010. <http://bittorrent.org/beps/bep_0005.html>.

[14] Payberah, Amir, and Seif Haridi. "Kademlia: A Peertopeer Information System Based on the XOR Metric." Class Lecture, 23 Mar 2009. Web. 18 Mar 2010.

[15] Menasche, Daniel S., Antonio A.A Rocha, Bin Li, Don Towsley, and Arun Venkataramani. "Content Availability and Bundling in Swarming Systems." in Proc. of CoNEXT, Dec 2009.

[16] Peterson, Ryan S., and Emin Gun Sirer. "Antfarm: Efficient Content Distribution with Managed Swarms." in Proc. of USENIX *NSDI*, 2009.

[17] György Dàn, Niklas Carlsson, Ilias Chatzidrossos. "Improving BitTorrent Performance using Peer Migration and Swarm Management,", Technical Report, IR-EE-LCN 2010:010, May 2010.

[18] Sundell, Jari. "The libTorrent and rTorrent Project." Web. 31 May 2010. <http://libtorrent.rakshasa.no/>.

[19] Norberg, Arvid, Ludvig Strigeus, and Greg Hazel. "Extension Protocol.", 28 Feb 2008. Web. 31 May 2010. <http://www.bittorrent.org/beps/bep_0010.html>.

[20] Wu, Di, Prithula Dhungel, Xiaojun Hei, Chao Zhang, and Keith W. Ross. "Understanding Peer Exchange in BitTorrent Systems." in Proc. of IEEE Peer-to-Peer Computing (P2P), 2010.

[21] Modelnet UCSD Systems and Networking, Web. 22 Sep 2010. <https://modelnet.sysnet.ucsd.edu/>.

[22] Rao, Ashwin, Arnaud Legout, and Walid Dabbous. "BitTorrent Experiments on Testbeds: A Study of the Impact of Network Latencies." INRIA France, in Proc. of JDIR, 2010.

[23] Engling, Dirk . "opentracker – An open and free BitTorrent tracker." Web. 19 Nov 2010. < http://erdgeist.org/arts/software/opentracker >

[24] Bharambe, Ashwin R., Cormac Herley, and Venkata N. Padmanabhan. "Analyzing and Improving a BitTorrent Network's Performance Mechanisms." in Proc. of IEEE INFOCOM, 2006.

[25] N.V. Smirnov "Tables for estimating the goodness of fit of empirical distributions", *Annals of Mathematical Statistic*, vol. 19, num. 2., pp. 279-281, 1948