MASTER THESIS

PeerSelector: A framework for grouping peers in a P2P system

Rakesh Kumar
Department Of Communication Systems
Royal Institute of Technology (KTH)
rakeshk@kth.se

June 19th, 2012

Master's Thesis at TSLab
Supervisor: Flutra Osmani (KTH)
Examiner: Björn Knutsson (KTH)

TRITA-ICT-EX-2012: 109

Abstract

This master thesis presents a framework called PeerSelector that has been designed and implemented to group peers in a Peer-to-Peer (P2P) system according to certain criteria. The framework is portable and can be deployed with any distributed P2P system. We devised the framework with such functionality in mind because we consider that grouping peers according to certain criteria can benefit the users of the system by providing them with more flexibility to group peers according to their own interests, without depending on entities such as ISPs for peer clustering.

We designed and implemented a modular architecture for the framework. More specifically, PeerSelector consists of modules that implement basic functionalities such as grouping peers according to geo-location, RTT-based latency, and the number of AS (Autonomous System) hops. When peers are grouped according to the respective metrics, they are stored in queues, namely the distance, latency, and hop-count queues. Any P2P system that is integrated with our framework fetches peers from such queues, on demand.

The results from the framework functionality testing show that the framework is successfully able to cluster peers according to the user's indicated interest. In addition, the framework has been integrated with two existing P2P protocols with minor adjustments, confirming the flexibility and portability of the framework across applications.

We have carried out experiments to investigate if using our peer clustering techniques helps a P2P client increase its download performance. In our experiments with a live swarm, we learned that grouping peers according to geo-location does not influence the download performance drastically: download performance increases slightly or remains the same for almost 75 percent of the cases. For the two other clustering metrics, latency and AS hops, our preliminary experimental results don't always show an improvement of the client's download performance.

Sammanfattning

Denna examensarbete presenterar en konstruktion kallad PeerSelector som har blivit designat och implementerat för att gruppera klienter i ett P2P-system efter vissa kriterier. Konstruktionen är portabel och kan användas med alla distribuerade P2P-system. PeerSelector är utformat så att användarna kan gruppera klienterna efter deras egna intressen, vilket ger ökad flexibilitet, utan att vara beroende av enheter så som ISP.

Vi har designat och använt en modulär arkitektur för konstruktionen av PeerSelector. Detta betyder att den består av moduler som utför basfunktioner så som att gruppera klienterna efter geografiskt läge, RTT-baserad fördröjning, och antal AS-stopp. När klienterna är grupperade efter respektive ämnesdomän, är de förpassade in i olika kösystem, i synnerhet efter sträcka, fördröjning och stopp-räkning. Alla P2P-system som är integrerade med vår konstruktion hämtar på begäran klienter från dessa kösystem.

Resultaten från funktionalitetstester visar att PeerSelector kan gruppera klienter efter användarens intressen. Vidare har denna konstruktion inkorporerats med två existerande P2P-protokoll genom mindre justeringar, vilket bekräftar såväl flexibiliteten och portabiliteten av konstruktionen.

Vi har genomfört experiment för att undersöka om användandet av våra klientgrupperande tekniker hjälper P2P klienter att öka deras nedladdningshastighet. Genom våra experiment lärde vi oss att grupperandet av klienter efter geografisk position inte nämnvärt påverkar nedladdningshastigheten, den ökar något eller förblir det samma i ca 75 % av fallen. I två andra mätningar, fördröjning och AS stopp, visade vårt första experimentella resultat inte alltid en förbättring av klientens nedladdningshastighet.

Acknowledgements

I would like to thank my examiner Björn Knutsson for his valuable advice and support during this project.

I am extremely grateful to my supervisor Flutra Osmani for her guidance and support. Her comments and suggestions helped me to improve the project. Without her insights and contribution this research project would not have been possible. The co-operation is much appreciated.

I thank Fu Tang for his support during the integration and testing the framework with Swift and my friends for their help in the translation of abstract into the Swedish Language. I would also like to acknowledge Niklas Wahlén of the *Swedish Institute of Computer Science SICS*, for providing the ASDistance database.

I would like to take this opportunity to thank my beloved parents for their patience. A special thanks to my sister Naweli and my wife Namrata for continuous help, support, faith and encouragement. I felt motivated every time I talked to them.

Table of Contents

List of Figures

List of Tables

List of Codes

1. Introduction

During the last decade, Internet traffic has had significant growth [9][10]. Peer-to-Peer (P2P) applications are among the major contributors to traffic growth [6][7][8][15]. In P2P systems, peers offer their available resources to other peers and get the needed resources in return. Examples of P2P communications are content mass distribution, online gaming and video telephony [2][3][4].

There is always a need to improve download performance of such P2P systems, because users always want the content to be downloaded in less time [6]. Additionally, Inter-ISP traffic generated due to P2P applications is high. The amount of traffic crossing ISP boundaries is a financial burden for ISPs [5][14]. Similarly, latency plays a major role in P2P applications like live streaming and online gaming [4]. How a peer selects its neighbor, matters in all these scenarios.

Researchers have already proposed several alternatives to solve the above three issues. For example, biased neighbor selection (BNS) technique was proposed by Bindal et al [18]. Using this technique, peer selects most of its neighbors from the local ISP. In this approach, AS-Hop is used as the metric for determining the locality of peers. There are two means of biasing neighbor selection: 1) tracker is modified to return local peers, and 2) a P2P traffic shaping device is placed at the ISP side, modifying the response from the tracker to the peer and supplying local peers.

Similar to the BNS technique, Biased Unchoking [1] technique was also proposed by Oechsner et al. In this approach, neighbors with better locality values are selected. A better locality value is determined using the AS-Hop count metric. However, these two approaches are proposed only for the BitTorrent system. They are not designed to support other P2P protocols. Furthermore, only the AS-Hop metric is used for grouping peers; other metrics like the geographic distance and latency are not considered.

Many of the solutions suggested for peer selection are centralized in nature. An extended set of work has been proposed for locality-awareness using oracles, such as [14] by Aggarwal et al. An oracle is a central tracker placed at each ISP; each ISP provides an oracle service to its users. To illustrate, P2P users send the list of possible neighbors to the oracle and the oracle ranks them according to certain metrics. P2P users use this sorted list to select neighbors. However, this approach requires the deployment of oracles on each ISP.

A decentralized solution that has been suggested is the CDN-based oracle project named Ono [16]. In this proposal, CDNs send the information about the replica server using dynamic DNS redirects. Such information is used to guess the position of the P2P client. If two clients receive the same set of replica servers, then they are likely to be close to each other [16].

Furthermore, an interesting strategy that has been proposed by Liu et al [5] for BitTorrent to localize P2P traffic within the ISP boundaries and improve download time embeds the locality awareness feature inside the BitTorrent implementation. Authors suggest three different ways of locality biasing. In the first approach, the tracker sorts all the available

peers in its list and returns the sorted list of peers. This is similar to the BNS policy. In second approach, a peer unchokes --- a temporary approval to upload --- 4 of its closest neighbors. By doing this, a peer prefers to exchange data with its nearby neighbors. Finally, in the third approach, a peer downloads pieces closest to itself. Their findings suggest that the last two approaches can help in reducing download time and the first approach reduces Inter-ISP traffic. However, these policies can only benefit BitTorrent users and can only reduce Inter-ISP traffic generated by a BitTorrent application. Like BNS and BU approaches, this proposal also considers only the AS-Hop metric for grouping peers.

For online gaming and for streaming applications, low latency is desirable. Several approaches have been proposed to reduce latency in P2P systems in the past. A new latency prediction system called Htrae [4] has been proposed for online gamers' matchmaking. It uses a combination of two classic approaches for latency prediction: geo-location information and a network coordinate system (NCS). Using this information, each machine is assigned a co-ordinate in virtual space. The distance between two machines in virtual space is used to predict Round Trip Time (RTT) between them.

Even though considerable amount of work is done in this field, none of these solutions are generic in nature. Many of the solutions presented in the past are designed only for the BitTorrent system. Moreover, they are more implementation specific and require that the locality awareness functionalities be embedded deep inside the implementation. Additionally, approaches like the oracle-based or the CDN-based approach depend on additionally deployed Internet infrastructure to function. Finally, many of the proposed mechanisms are applicable only at the ISP side, thus providing no flexibility to end-users to group neighbors based on their own requirements.

To address the above concerns we propose, implement and evaluate a more generic approach, which can be deployed at the end-users' side and can be ported across different distributed applications. In this thesis work, we attempt to provide flexibility to the end-users to cluster peers based on their own needs and interests.

1.1 Goal and objectives

Our *goal* is to create a framework that integrates previous approaches, namely locality-biasing, low-latency, and nearby-AS proposals into one flexible and portable mechanism that can be deployed with other P2P systems and does not depend on external entities or additional Internet infrastructure to function.

To achieve our goal:

1. We investigate locality-biasing, low-latency, and nearby-AS proposals for P2P systems and identify useful approaches. Then, we group these approaches --- the geographic distance, low-latency and the AS-Hop count approaches into a set of policies for our framework.

2. We design a framework, which
    - Is portable and could be integrated with different P2P systems

- Provides flexibility to the end users to cluster peers according to their preferences
- Provides a modular architecture so that the new policies can be easily implemented with the already existing policies

3. We integrate the framework with at least one already deployed P2P system to evaluate its functionalities.
4. We perform an experimental evaluation of the integration of PeerSelector with a deployed BitTorrent P2P system.

1.2 Contributions

The framework is *designed* to cluster peers. The framework supports grouping of peers according to geo-location. To get the information about peers' geographic location, the GeoIP database was used. The framework loads the GeoIPASNum and the GeoIPCity database from Maxmind into memory [34]. As of 4th April 2012, the database has 3,448,149,321 IP addresses from 250 different countries. According to Maxmind [44], the database is 99.8% accurate at the country level. The database contains geographic information such as longitude, latitude, ASN, city, country, and continent information for IP addresses. With the help of GeoIP APIs, this information can be retrieved and used to compute a weighted score (will be described later) for each peer.

To group peers according to latency, our framework uses two techniques: active probing and history prioritization. In active probing, a mean value of three iterative probes --- issued at an interval of 200 ms --- was used. The history prioritization technique is used for calculating latency more quickly and without any overhead by using previously computed RTT values. Previously stored RTT values can be a good estimation for future RTT values. According to Agrawal et al [4], around 95% of the nodes in their database, over the period of 50 days, exhibited RTT values with coefficient of variation under .2. This indicates that RTT in the Internet is quite stable and history prioritization technique can save time and CPU.

To distinguish local peers from remote peers, the AS-Hop count was used. Peers from the same AS are considered local and peers from the outer ASes are considered remote. Instead of constructing an IP level map and an AS level map as described by Liu et al [5], we used the ASDistance database created at the *Swedish Institute of Computer Science SICS*, which uses BGP routing tables from the "Route Views Project" [45].

After determining which peers are local, they are then stored in the main database of PeerSelector and can be returned as a list to any P2P application, on demand. We used MySQL++ database for storing peers information [31]; MySQL++ is the C++ API for MySQL.

Furthermore, to distinguish peers from each other we have defined a set of metrics. They are: the score, RTT-based latency, and the AS-Hop count metrics. Using these metrics, we discover closer peers. Closer peers can be divided into three categories: geographically at closer (less) distance, lower latency or less AS-Hop count. These three metrics are imprinted in three corresponding policies, labeled by the same names. In addition to these three policies, a random policy was also constructed to randomly group peers, without biasing for locality.

After obtaining peer's information for any given policy and after determining peer properties, we group those peers. The process is known as peer profiling. For peer profiling we use queues, which are created on request. There are four different queues --- random, distance, latency and hop count queues --- that store peers, profiled by the four different policies listed above. Queues are created by running SQL queries on database's tables. Currently, four tables: infohash_ipaddr, score_table, rtt_table and ashop_table are created, however, PeerSelector's modular architecture allows us to add more policies and tables easily.

1.3 Findings

Experimental results highlight the *"potential" benefit* of using the PeerSelector framework. The framework was integrated with Swift and BitTorrent. So, the interface provided by PeerSelector is generic in nature. The framework provides the flexibility to the user to bias neighbors according to their indicated interest. With the current version completed in this thesis work, four types of user interest are tested: random, less distance, low latency and less hop count. The framework is able to cluster peers according to the user's shown interest.

After making sure that the framework worked correctly, it was integrated with P2P systems. First, the framework was integrated with Swift P2P protocol and tested in a manually controlled setup in PlanetLab [54]. When integrated with Swift, we have seen that less distance and low latency policies performed better than random policy. It's also been observed that when the RTT-based latency metric was selected, content download started with a 25-30sec delay, compared to other policies. This is known as the learning period, during which, our peer is actively probing peers in the swarm and calculating the mean RTT to each peer.

On the other hand, when the framework was integrated with BitTorrent, experimental results on download performance were not as conclusive. In one trial, we have seen the random policy performing better than all three policies, however, we have seen it performing worst in the next trial. The behavior of the score metric seems to be slightly more consistent as it provided on average consistent and better download speed in all scenarios.

There seems to be a good improvement in terms of download speed when framework is tested with manually configured set up in PlanetLab with Swift. In case of BitTorrent, even though obtained results are not so promising in terms of download speed, this project is a first approach of integrating all peer profiling mechanisms in one framework, which could be deployed with any distributed P2P system, and where users can specify their preferred interest. In addition, framework does not require any external entity or additional Internet infrastructure support like in oracle, P4P or CDN based oracle approaches.

Furthermore, the framework is scalable. Theoretically, the size of PeerSelector database depends upon the operating systems drive file system where Mysql is installed. Scalability testing was not done for this framework however, during the testing phase, database table size grew up to 500 entries. We have observed that if the number of peers returned by the tracker

is more than 30 then RTT-based latency metric and AS-Hop metric have experienced a longer learning period.

1.4 Scope

The scope of this thesis is to develop a flexible framework to group peers according to locality, latency and nearby ASes. Framework should be portable and should be deployed with any P2P systems. Under the current scope of this project, the framework will be integrated with Swift and BitTorrent P2P protocols.

1.5 Audience

Project will benefit the end users by providing them the flexibility to group peers according to their preference. Since the framework is portable, it will enable other P2P systems like BitTorrent, Swift, DHT to use locality-biasing feature without the need additional Internet infrastructure support.

1.6 Structure

The report is organized as follows: Chapter 2 discusses the background and Chapter 3 describes the related work. Chapter 4 explains the methodology, tools and libraries used during the project to achieve the goal. Chapter 5 describes the detailed design and Chapter 6 focuses on its implementation. Chapter 7 covers the experimental set up and the evaluation of the projects and results are compared using various graphs. Chapter 8 is about the discussion of results and problems uncovered during the design and implementation of project. Chapter 9 is the conclusion of this dissertation. Chapter 10 discusses the work that should be carried out in future. Appendix A depicts design diagrams and Appendix B is the source code. Appendix D and E contain the acronyms and the glossary, while appendix F contains the implementation details.

2. Background

In this chapter, knowledge required in understanding the design, implementation, and usage of the framework will be presented. In the first section, basic concepts of P2P networks are described. In the second section, overview of three P2P protocols: Swift, BitTorrent and DHT will be presented. In the next section, the need for locality biasing will be explained.

2.1 Peer-to-Peer networks

Peer-to-peer networks are communication networks where peers exchange information without the need of a central entity. Every peer in the network can act both as a client and as a server. In other words, peers are both suppliers and consumers of resources. Peer to peer systems started with the music file sharing system called Napster [51]. Then Gnutella [47] and Kazaa [52] came into the market [46]. Now, for content mass distribution, BitTorrent [3] is predominantly used [15]. Moreover, P2P is also used in multimedia communication networks.

According to R.Schollmeier [26], a peer-to-peer network is a distributed network, where participants share a part of their hardware, for example CPU, memory, and network link capacity. These shared resources are required to provide P2P services like file-sharing.

P2P networks can be classified into two categories: Hybrid P2P and pure P2P. In hybrid P2P systems, a central entity is involved that is contacted first for obtaining meta-data or for verifying security credentials. In pure P2P systems, there does not exist a central entity.

2.2 Peer-to-Peer protocols

2.2.1 Swift

Swift is a new multiparty transport protocol designed for content distribution and retrieval in P2P systems [20]. It can be used for file download, Video on Demand (VoD) and live streaming. Protocol is designed to store and deliver data over the Internet using the unique identifier called root hash. It is calculated recursively from the content. Although Swift is designed as a transport protocol, it mostly runs over UDP. Since it is a generic protocol, it can also run over TCP, HTTP or as an RTP profile.

In Swift, information is exchanged using datagram. Datagram is a sequence of messages that is forwarded as a unit to underlying protocol i.e. messages are multiplexed and sent across the network in datagram. When Swift is used over UDP, it contains a type field and the message payload length field, where type field represents the type of the message and payload field contains the actual message, depending on the type field.

According to the Master's thesis by G.C.Anon on "Joining BitTorrent and Swift to improve P2P transfers" [21], Swift is able to carry almost the entire Internet traffic. Swift RFC [20] says that Swift can work behind NAT and also works well behind the firewall. It also allows

the user to choose the congestion control algorithm according to his requirements. Another design goal of Swift is to reduce warm-up time.

Swift introduces the concept of the channel to distinguish different file transfers between two end-points. Channels are identified using a 4-byte channel number and it is prefixed at the start of each datagram.

2.2.1.1 Swift operation

Swift operation can mainly be divided into three processes. First, a peer joins the swarm, which consists of a group of peers sharing the same content. Once the peer joins the swarm, it starts retrieving pieces of content and then, it leaves the swarm [20].

Joining the swarm

When a peer wants to join a particular swarm, it first registers itself with the tracker. It then gets the IP address and port number of the peers already in the swarm. This is valid in case of a centralized tracking architecture. In case of decentralized tracking mechanism, Peer Address Exchange (PEX) gossiping is used to get the list of peers already in the swarm. PEX gossiping is described in next section. To join the swarm, a peer should have the swarm ID. Each swarm is identified using a unique identifier called a swarm ID. In the downloading and the VoD scenario, swarm ID is a root hash while in the live streaming scenario this is a public key. First, a peer sends a datagram containing the HANDSHAKE message to all the peers in the swarm. Peers reply to this with a datagram containing a HANDSHAKE message and HAVE messages. HAVE messages convey chunk availability.

Distribution of content

After receiving the datagram containing the HANDSHAKE and HAVE messages, the peer (requester) sends a datagram containing HINT message asking for the chunks that it wants to download. Other peers in the swarm may respond with a datagram containing HASH, HAVE, and DATA messages. Hashes inside the HASH message are used by the requester to verify content, HAVE message is used to convey the list of chunks that other peers have, and the DATA message contains the actual requested chunk in the datagram containing HINT message. After receiving the chunk, the requester acknowledges it with a datagram containing an ACK message. It also sends datagram to all its peers, containing a HAVE message, for the chunk it has received. It then sends a datagram, containing a HINT message, requesting for new chunks.

Leaving the swarm

There are two ways of leaving the swarm. When a peer wants to leave explicitly, it sends an explicit leave message [20] to peers in the swarm and leaves the swarm. In case of implicit leave, peers stop responding to messages. Swift Operation is explained pictorially under the heading “Swift on the wire” by Baker et al [48].

2.2.1.2 Swift decentralized tracking

Swift supports a tracker-less download using PEX gossiping algorithm or using DHT. In this section, PEX gossiping will be discussed. DHT will be covered in section 2.2.2.1. Swift uses Peer Address Exchange (PEX) algorithm for finding peers. Peer addresses are exchanged in a gossiping fashion. Peers interested in other peer addresses send PEX_REQ messages. After receiving the PEX_REQ message, a receiving peer may reply with the address of the peers it has recently exchanged messages with, using a PEX_ADD message.

2.2.2 BitTorrent

Bittorrent [3] [23] is among the most widely used P2P protocol for content distribution [22]. If a file has to be transferred using BitTorrent protocol, it is first divided into pieces. Each piece is then encrypted using a cryptographic hash. Each peer involved in downloading, downloads these pieces and also makes these pieces available to other peers in the swarm. Pieces are downloaded unordered and are ordered by the BitTorrent client.

BitTorrent client is a software program that uses BitTorrent Protocol for upload and downloads [23]. It manages all the pieces that the peers have already downloaded, pieces that the peers can upload, and the pieces that peers want to download. There are lots of options available when it comes to choosing the BitTorrent client [24]. Few examples are µTorrent [53] and Vuze [17]. Once the user launches the client and provides the torrent file to it, it starts the download. In this project, we have modified the client available with libtorrent (Rasterbar) library [13]. The reason for such modification is to integrate PeerSelector functionalities with BitTorrent.

In order for peers to locate each other, BitTorrent provides two mechanisms: tracker based centralized approach or DHT based decentralized approach. The DHT based peer selection mechanism will be explained in later subsections.

Tracker based approach uses a torrent file for finding peer's information. It is distributed either through websites or through central repositories. The torrent file contains information about the file to be downloaded, like file size and hashing information. The file also contains the URL of the tracker. Tracker is a central entity that assists P2P file transfers. Tracker contains the IP address and the port number of all the peers in the swarm. They help peers to discover each other.

In BitTorrent terminology, a downloader who does not have 100% of the content is called a leech. When a peer successfully downloads all the pieces of a particular file it becomes a seed. So, seeds are basically the peers with all the pieces.

2.2.2.1 BitTorrent operation

A downloader first downloads the torrent file from any web server. The torrent file contains metadata about the file to be downloaded and information about the tracker, which keeps track of all the peers involved in the download of a particular file. When the peer (requester) contacts the tracker using the torrent file, it gets the address of initial set of peers already involved in the download. The peer then initiates the connection with these peers and starts bartering for pieces. Few pieces are obtained for free. This is called optimistic unchoking.

Choking algorithm

It is the algorithm used by the BitTorrent peer to decide which peers to upload to and which not to upload to. Using the choking algorithm, a peer builds its neighbor set whom it is willing to upload data to. These neighbors are called *unchoked peers*.

Algorithm works in two modes depending on the peer type. If peer is a seeder, then peer keeps the three most recently unchoked peers as unchoked peers. However, if the peer is a leecher, then it unchokes the three best peers from which it is getting the best downloading speed. This strategy is called *tit-for-tat*. The choking decision is made every 10 seconds. Choking decision only stops the uploading, downloading can still be in progress. Furthermore, the connection does not need to be renegotiated once the chocked peer is unchoked. In both modes, every 30 seconds, a new peer from the list of chocked peers is unchoked and the peer with longest unchoked time is choked. This approach is called *Optimistic Unchoking*. Optimistic unchoking method is devised to check if the currently unused links are better than the one currently being used in download.

2.2.2.2 DHT

DHT --- Distributed Hash table is used to store (key, value) pairs of the node. Each node has a routing table containing contact information about some nodes near to the own node. A node in DHT is identified using Node ID. These nodes collectively form the distributed system without using any central entity for co-ordination.

BitTorrent uses DHT for trackerless download. If DHT is used, then each node becomes the tracker as it has information about the nearby peers. In order to start the download, node searches the ID corresponding to the infohash of the torrent in its own routing table. It then contacts the closest nodes and asks for peers it knows about. If the contacted node knows peers with the given infohash, it returns those peers; otherwise, it starts searching its own routing table and returns the nodes closest to the infohash of the torrent. The original node (requester) iteratively queries nodes closest to the infohash of the torrent until and unless it finds any closer node. Refer to the Mainline DHT specification [25] for detailed information.

2.3 Why locality biasing?

P2P networks are created over the actual physical network. This overlay network is unaware of the actual network topology. P2P applications often create challenges for ISPs because most P2P systems rely on the application layer routing on the overlay topology and not on the Internet based routing [14]. There is also a challenge for P2P systems to construct an optimal overlay, because P2P systems are agnostic of the underlay. To overcome these problems, biased P2P overlay construction is required.

Most of the locality-biasing approaches are based on the translation of the peer's information from overlay network to physical network and vice-versa. In other words, in locality-awareness approaches, peers are selected from physical network's point of view and not from the overlay perspective.

3. Related work

In the related work chapter, we present different approaches for locality biasing proposed in the past, their benefits, and their limitations. In the first section, common approaches for modifying the peer's neighbor set using BNS and BU technique are explained. In the next section, different architectures which require changes at ISPs side are described. Third section describes about the client side locality awareness techniques for BitTorrent clients. In the fourth section, a latency prediction technique *Htrae* is explained and finally, in the last section, impacts of locality biasing are discussed.

3.1 Neighbor selection

In order to improve the download/upload performance of BitTorrent, researchers have proposed solutions to select a better active neighbor set. Different solutions have different ways of defining which active neighbor set should each peer select.

Zhang et al [40] have proposed "soft-worst-neighbor-chocking" algorithm. In this algorithm, instead of applying the tit-for-tat policy, peer chokes one of its active neighbors based on the probabilities which are exponentially weighted. Additionally, similar to optimistic unchoking approach, peer unchokes a new neighbor randomly. However, this solution allows peers to stick to a better performing neighbor for a longer period, which in turn allows the system to stay in a better performing mode for longer time. Nevertheless, in the proposed optimal neighbor selection approach, neighbors are selected randomly.

Another approach for neighbor selection, which is presented by Bindal et al [18] is the biased neighbor selection (BNS). In this approach, neighbor sets of peers are modified according to the proximity, in terms of location or AS-Hops. BNS can be applied to the ONO [16] suggested peers or with the peers returned by the oracle service. There are two ways of implementing BNS [18]: 1) modifying the tracker and client, and 2) using a P2P traffic shaping device at the ISP side.

In the first implementation, a tracker is modified to return peers from the same AS. Tracker fills the neighbor set with 35-k internal neighbor and k external peers. Value of k is implementation specific. Client can also contact the tracker for more local peers if the number of local peers is less than 35-k. Bindal et al [18] also suggests including a new "X-Topology-locality" tag, which is used to identify peers from the same ISP.

In the second approach, a P2P traffic shaping device is used for modifying the neighbor set. These traffic shaping devices are situated at the edge routes of ISPs. These devices keep track of all the peers inside the ISPs downloading the same content. When a new peer wants to download the same file it contacts the tracker to get the list of peers. These traffic shaping devices intercept the response from the tracker to the peer. Then they modify the peer list and replace external peers with internal peers.

Similar to BNS, Biased Unchoking (BU) proposed by Oechsner et al [1] is another method for neighbor selection. In contrast to BNS, in the BU approach the neighbor set is not modified but the choking algorithm is modified to unchoke peers from the preferred list. Both BNS and BU need the underlay information to the overlay to construct the P2P overlay. This information is given by information servers such as iTracker [19] or SmoothIT information service [41].

In the paper by Oechsner et al [1], for the performance evaluation, locality value L(x,y) is obtained from an information server. The AS-hop metric is used for calculating locality value L(x,y), where L(x,y) represents the number of AS-hops between peer at address x and peer at address y. Peers with L(x,y) <= T are defined as preferred peers and the rest are called non-preferred peers. If T is 0, then only peers from the same AS are in the set of preferred peers. With BU, peers from the preferred set are chosen first for optimistic unchoking, if the preferred set is empty i.e. local peers are not available, then the peers are selected from the non-preferred set.

The BU approach gives better results when load in the swarm is high. In a scenario where large fraction of peers resides in the same AS, BNS can reduce inter-AS traffic up to 20-30%. However, a combination of BNS and BU can reduce traffic up to 80%. BU, and BNS and BU together can reduce inter-AS traffic only when the mean seeding time is less and the load on the swarm is high. If a very small fraction of peers resides in the same AS, then the combination of BNS & BU is considered a better choice than the individual selection. Authors suggest that both mechanisms complement each other and should be used together to get the best results. When they are used together, local peers are obtained using BNS and then they are unchoked using BU.

However, focus of the work in research paper by Bindal et al [1] and Oechsner et al [18] is to benefit ISPs and the user's QoS is not improved. In addition, these two proposed approaches are suggested for BitTorrent. Moreover, it is required to embed these functionalities deeper into the BitTorrent implementation.

3.2 Architecture for neighbor biasing at ISP side

3.2.1 P4P architecture

Xie et al [19] introduces a new architecture called P4P, which could benefit the ISPs as well as users. Paper introduces the concept of iTrackers and appTrackers. iTrackers act as provider portal and each network provider has to maintain its own iTracker for its network. iTracker maintains information about the peers such as distance between two peers in terms of AS-Hop count. On the other hand, appTracker is integrated at the client side. In case of tracker-based application, appTracker queries iTracker and gets the necessary information required for neighbor selection and relays the information to the interested peer. In case of tracker-less applications, peers directly contact the iTracker.

The architecture requires changes at the application side and also at the network provider side. It is still uncertain if applications and network providers will adopt this new architecture or not. Furthermore, as the implementation presented in the paper is application specific so is the interface, which has to be changed for new P2P applications.

3.2.2 Oracle based architecture

In the architecture presented by Agrawal et al [14], each ISP provides an oracle to P2P users. It is a central tracker placed at each ISP, which maintains information about all the customers. P2P users supply their neighbor sets to the oracle and oracle rearranges them according to certain metrics. The oracle may rank peers according to the number of AS hops, bandwidth or distance to the edge of AS, among others. So, the peer recommendation is done at the ISP side. Once the recommendation is made, P2P node can use this information to select its neighbors. So, the P2P users do not have to involve themselves in calculation as they can rely on ISPs. Benefit from the ISP point of view is that they can do traffic engineering in a better way as they can influence the peer selection process. Additionally, it is available for all overlay networks.

However, this approach requires large infrastructure to be in place. It requires deployment of oracles on each ISP. It also requires ISPs and their users to cooperate and to trust each other. The service is not yet offered by the ISPs.

3.2.3 CDN-based oracle architecture and other techniques

Another scalable technique proposed in the CDN-based oracle project named Ono is lightweight and does not require any new infrastructure [16]. It does not depend on cooperation between ISPs and their users. CDNs send the information about the replica server using dynamic DNS redirects. Ono project uses this information to guess the position of the client. If two clients exhibit similar redirection behavior, then they are likely to be close to corresponding replica server and thus close to each other [16].

Unlike other traffic engineering approaches at ISP side, like deep packet inspection or placing a cache at ISP's gateway[43], this technique does not has to deal with any legal issues. As of January 2008, this solution was deployed over 3000 networks with over 120,000 subscribers. The results collected from the deployment show that over 33% of the time CDN-based oracle technique select neighbors within the same AS. Neighbors selected with this approach have experienced two times lower latency than the ones selected using random policy. Solution also provided better average download rates.

However, the project only biases neighbors based on similar redirection information. Other aspects of the network, like bandwidth of connection, are not considered due to the limitation of the Azureus client. Unlike oracle based approaches, Ono project implementation is specific to BitTorrent protocol as it is a plugin to Azureus/Vuze [17] client. It does not support other peer-to-peer protocols. Furthermore, since the behavior of Ono depends on the behavior of CDNs, any change in CDNs' behavior might affect the behavior of Ono. However, authors claim that there is no need to change Ono's implementation, even if the behavior of CDNs changes.

HTTP-based P2P protocol proposed by Shen et al introduces a technique called *HTTPifying* [43]. The idea is to use the web cache proxies deployed at ISPs for storing P2P traffic.

Results show that *HTTPifying* caching techniques is helpful in reducing the P2P traffic on transit links and on the backbone.

3.3 Locality awareness techniques for BitTorrent

In the context of locality awareness for BitTorrent, many techniques have been proposed. Work in paper by Liu et al [5] suggests embedding of the locality awareness technique inside P2P systems. Liu et al created a detailed AS level map for different policies. In optimal policy, a minimum spanning tree is constructed, where each node represents a peer and an edge weight represents the AS-Hop count. Problem with this strategy is that each node downloads all its content from its only parent that is at the minimum hop count. Few other drawbacks of this approach are mentioned in section III B of the paper. Authors also suggest embedding the locality awareness mechanism inside BitTorrent. This is done in three different ways: tracker locality, chocker locality, and piece picker locality.

In tracker locality approach, tracker sorts all the peers in the swarm in ascending order according to AS-Hop count. When a request from a peer arrives, tracker sends the sorted list and the AS-Hop count --- from the requesting peer to all other peers in the list. Secondly, chocker locality approach is based on unchocking 4 best neighbors. Best neighbors are decided according to AS-Hop count. Finally, in piece picker locality technique, a distance value is assigned to each piece. Distance value is calculated from AS-Hop count value. The piece with the lowest distance value is preferred.

Results show that the chocker and piece picker locality awareness techniques are good from user's point of view as they reduce download time. On the other hand, tracker policy benefits ISPs by reducing inter-ISP traffic.

However, the above approach embeds locality awareness feature in the implementation according to the AS-Hop count only. Latency to each peer and the geographic distance are not considered. Moreover, the approach is not portable as it is BitTorrent specific.

3.4 Latency prediction

By Agrawal et al [4], a new latency prediction system called *Htrae* is proposed. System allows peers to cluster themselves in way so that they have low latency to each other. It uses a combination of two classic approaches for latency prediction. One approach is to use the geo-location information and the other approach it to use network coordinate system (NCS).

When a peer wants to join the system, it first gets the actual coordinate of the node using GeoIP database [34]. This is called geographic bootstrapping. After that, whenever a peer determines its RTT to another machine, its coordinates are dynamically adjusted on the virtual space. If the RTT to the machine is low, then a virtual force is applied to the coordinate towards the machine and the coordinate is shifted towards the machine. Similarly, if the RTT to the machine is high, the virtual force is applied away from the machine. The original NCS system was not using the actual longitude and latitude information as used by Agrawal et al, for geographic bootstrapping. System was using fixed set of nodes called *landmarks* to find the virtual coordinates. Like NCS, standalone geo-location techniques had

also some limitations. The drawback of the geo-location prediction technique is if it inaccurately predicts latency to any machine, then that machine will consistently give poor performance to the player. Other drawbacks are also mentioned in [4]. By combining two approaches we get the benefit of both. In our framework, we have used history prioritization technique --- using previously stored RTTs, proposed in Htrea.

3.5 Impact of locality biasing

Several studies have been done in order to understand the impact of locality biasing on ISP transit traffic and end-user download rates [38][39]. Although the impact seems to be straightforward and the construction of peer-to-peer overlays using locality information seems to decrease transit traffic and should increase download speed, in reality, results are quite diversified.

The impact of locality biasing was explored by Cuevas Rumin et al [39], and a case study about the implication of locality biasing, considering the demographics of torrent and speed of different ISPs are presented. The paper describes the impact of locality biasing in two different policies.

The first policy is named Local Only If Faster (LOIF), where local peers are preferred over remote peers only if they are faster. However, in the second policy, named Strict, all remote peers, irrespective of their speed, are exchanged with local peers. If some remote peers are still left in the neighbor set then they are discarded, except one peer. Study was done on three largest ISPs from US and three largest ISPs from Europe. According to the study, for fast ISPs, LOIF reduces transit traffic by 32% compared to random peer selection strategy. For slower ISPs, transit traffic reduction is 10% when compared with random policy. Additionally, end-user's QoS is preserved in the LOIF case.

In the case of second policy, effects of locality biasing in terms of reducing transit traffic are large. For fast ISPs, transit traffic can be reduced by 55% by selecting the local peers, and in case of slower ISP this number is 39%. The penalty at the end- user side, in terms of download speed, is less than 6%.

According to the research paper by Cuevas Rumin et al [39], transit traffic can be reduced up to 96-97% by limiting the number of inter-AS overlay links. However, user's QoS drops by 18% in case of EU (slower) ISPs and 3% in case of US (fast) ISPs. Nodes on "unlocalizable" torrents pay a heavy penalty of 99% in this case. Torrents with one or few nodes in one ISP are called *"unlocalized"* torrents. The paper also claims that when ISPs are on dense mode then the gain due to locality biasing is higher. To evaluate the impact of locality on inter ISP connections, Le Blond et al performed a large scale experiment with hundreds of thousands of peers [42]. With the high locality value, author reported a 50% decrease in the inter-ISP traffic. Their finding also revealed that the fast initial seeds are important to achieve fast download and low inter-ISP traffic.

In general we can say that locality biasing has a great potential and it is an effective technique in reducing the transit traffic. However, the results presented in several research papers are not so promising from user's point of view.

4. Methodology

This section describes the approach used during the project, design model, the tools used for designing, programming languages, databases, various libraries, and the methodology for experiments. The first section describes the procedure followed to gather and understand the requirements. Then the next section explains the design model and tools used to visualize the design. The third section gives information about the programming languages used and the mechanism built to integrate different languages. The fourth section explains different databases used for developing the framework. Then in the fifth section, we mention the external libraries used in the project and finally, in the last section, methodology for the experiment is described.

4.1 Procedure

In order to design PeerSelector, a detailed understanding of the P2P protocol and its working mechanism was required. After understanding the working mechanism, it was required to have a deeper insight into their neighbor selection mechanism and how that can be improved. To start with, we started understanding Swift Protocol Specification [20]. Then we started understanding how Swift obtains a list of peers that have the requested content. We found that it uses PEX gossiping algorithm to obtain the list of peers; additionally, it can also get the peer list using a DHT or a central tracker.

At this point we considered it necessary to develop a test application that takes as input a list of peers having the specific content and returns as output a list of peers sorted by city and country. Since a DHT implementation called *pymdht* [11] code was available to us and a very good support from the developer was available, we decided to use the *pymdht* implementation of the DHT to obtain the list of peers for a given infohash. Refer Appendix B.1 for more details.

The test application was developed based on the 12.2.3 Version of pymdht protocol [11]. Initial set of interface APIs were written in such a way that it can easily be extended for other P2P systems. For parsing the pymdht output, a parser listed in Appendix B.2 was written. *CalculateScore* API was written at this time, which was modified later. Test application's code snippet can be found at Appendix B.3 and B.4

Furthermore, one of the evaluation objectives of the project was to integrate PeerSelector with at least one already deployed P2P protocol. We chose BitTorrent for that purpose and started getting familiar with the BitTorrent protocol specification [12]. In order to test our framework with BitTorrent, we integrated libtorrent variant of the protocol with PeerSelector. We handed over the initial set of peers returned by the BitTorrent tracker to PeerSelector, sorted them, and then retrieved content from the sorted peers. For implementation, a C++ BitTorrent library - libtorrent-rasterbar-0.15.9 was used (explained later in this chapter).

After understanding Swift, DHT, and BitTorrent, we started designing a framework that is generic and could be integrated with different P2P systems, with less modification. Another important requirement was to provide the flexibility to the user to cluster peers according to their own interests.

To design a modular framework, visualization of the design was required. We have used Unified Markup Language (UML) for visualizing our design. Design model is explained in the next section.

4.2 Design model

Since UML is the de facto standard for the Object Oriented analysis and design [27] and, so far, it is the most widely used method of visualizing and documenting the software model [28], we choose this as a tool for designing our software framework.

The process of gathering and analyzing software requirements and converting all of the requirements into design became easy after using UML. According to the requirements, framework was first divided into three different components: core components, database components and queues. In addition, we intended to develop a framework, which can provide an interface that is well suited for other P2P systems. UML component diagram shown in Figure 5 helped us in achieving this. To define a PeerSelector class we have used the UML class diagram. Listing of attributes and methods required to define the behavior of PeerSelector became easy after that. For understanding the workflow of each method, sequence diagrams were used.

Diagrams are generated using GTK+ based free open source drawing software [29]. Its modular design with different shape packages and its easy-to-export feature, in different formats like PNG and PDF, inspired us to use it in the design phase.

4.3 Programming language

Given that the requirement of the project is to develop an application that is modular and easily extendible, it was required for us to choose any object oriented programming language. It was required for us to develop an application that can easily be plugged with different distributed applications, but primarily with Swift and BitTorrent transport protocols, which are written in C++.

After few brainstorming sessions we concluded that C++ is the preferred programming language to meet all our requirements. Java was considered as one of the programming alternative during the design phase, but it was ruled out because of the integration challenges with Swift and the complexity of creating a class which can be instantiated inside the libtorrent (Rasterbar) library.

However, during the implementation of AS-Hop policy, Java was used because the ASDistance database, which is used for finding AS-Hop count, was a compressed serialized Java object. In particular, we used the already existing database [30] for our AS-Hops policy. It is used in our framework for calculating the distance in hops between two ASes [30]. Package includes the ASDistance serialized class, which contains information about the distance between the transit ASes. In order to use this package, we had to instantiate the ASDistance class inside our application. During this phase of the project, we used Java and shell script to invoke the methods of ASDistance class. By invoking *getInstance()* method of the ASDistance class, we were able to calculate the distance in Hops between any two given ASes. Refer to Listing 6.7 for source code.

In addition to the C++ and Java, this project also uses bash script. This allows the framework to be extended easily because the bash script is pretty powerful. It allows us to execute external commands from the C++ code. It also gave us the flexibility to do some startup configuration without doing complicated coding. For example, in Listing B.5, *initialize.sh* file was called inside constructor to set the class path and to compile *ASHop.java* file. In addition, because of bash script we were able to avoid complex C++ coding for generating a C++ serialized object from the routing tables available from the "Route views project" [45].

4.4 Storage

4.4.1 MySQL++ database

Next task was to find proper storage for profiling that could easily be accessed from the core component written in C++. For the purpose of this project, MySQL++ was chosen, as it allows us to create tables and manipulate data inside the database from C++ code. MySQL++ provides C++ access APIs for accessing the MySQL database. In this project, MySQL++-1.7 version was used [31] [32].

Flexibility of MySQL++ allows us to easily create a connection object inside C++ code that can communicate with MySQL server --- running on local machine --- and manage the connection. It allows the framework to create query object and execute various SQL queries to store and retrieve peer information from the database. Because of the ease of use, SQL complex queries can also be run on the database to find some interesting patterns, such as: which AS is giving most peers with low latency or low score.

Before choosing MySQL++, Kyoto Cabinet was also tried [33]. The database used in Kyoto cabinet library is a data file with (key, value) pairs. Every key and value can be either of type string or of type binary. (Key, value) pairs are organized in the hash table or B+ tree.

Concept of datatable and datatype are not introduced in the Kyoto Cabinet database. Our requirement was to use datatables with multiple columns for information such as IP address, port number, hash and ASN. In addition, in Kyoto cabinet each key was associated with one value and should be unique inside the table. However, our requirement was to use composite keys composed of three columns: a hash, an IP address and port number, as primary key. An additional requirement was to have a separate table for each policy. Each table should have multiple fields and the relation between fields and tables was complex.

4.4.2 Other databases

The framework supports grouping of peers according to geo-location. Therefore, it was required to find the location of peers and group the peers according to the geographic location such as city, country and continent. To get such information, Maxmind GeoIP database was used [34]. As of April 4th 2012, the database has 3,448,149,321 IP addresses from 250 different countries. According to Maxmind [44], the database is 99.8% accurate at country level. Since the database is large and accurate, we have used it for the purpose of this project.

Furthermore, the database also provides the set of APIs to access the location information. As an example, GeoIP_record_by_addr API gives the longitude and latitude information.

The framework first loads GeoIPASNum and GeoIPCity databases from Maxmind into memory [34]. To get the AS number corresponding to the IP address, GeoIPASNum database was used. Longitude and latitude information and other geographic information --- city, country and continent information --- are available in the GeoIPCity database. With the help of GeoIP APIs, longitude and latitude information was retrieved. After getting the longitude and latitude information, the geographic distance between two IP addresses was calculated. The same API can be used to get the city, country and continent information. To get the ASN information, *GeoIP_name_by_addr* can be used [34]. Other useful APIs are *GeoIP_id_by_addr, GeoIP_org_by_addr*. Detailed information about the API set can be found at [50].

Another requirement was to find the AS distance between two peers. This is required to support AS-Hop policy. To find the distance between two peers in terms of AS hops, the ASDistance database was used. It is created at the Swedish Institute of Computer Science SICS and it uses BGP routing tables available from the "Route View Project" [45].

4.5 Libraries

libtorrent-rasterbar-0.15.9 - To test the functionalities of the framework with the already deployed BitTorrent P2P system it was required to change the source code of libtorrent's library. For implementation details, refer to section 6.4. After careful consideration, we decided to use libtorrent-rasterbar [13] because of many reasons. Firstly, it is written in C++ and can easily be integrated with our PeerSelector application written in C++. Secondly, library interface was very well documented. The API documentation is well structured and can be a very good start to get an insight of BitTorrent's implementation. Furthermore, library is an open source which is easy to use, CPU and memory efficient [13].

libboost-dev-1.40 - This library is a prerequisite for building libtorrent-rasterbar-0.15.9 library. We have used version 1.40 of boost library.

libcurl - version 7.19.7 - One of the implementation requirements was to find the external IP address of the system running our PeerSelector application. It is required to do the calculation needed for peer preference. It is especially important in a scenario where computer is behind the NAT. libcurl is used in this project, for this purpose. The library is open source software, used for transferring data to and from the server. In our project, curl is used to get data from the webpage which displays the external IP address [49] and then awk is used to parse the output. The curl tool by default displays data on the terminal.

4.6 Preliminary evaluation of the framework

In this section we describe our methodology for experiments. The framework was preliminarily evaluated with two P2P systems: BitTorrent and Swift. The evaluation was done in two phases: 1) functionalities of PeerSelector framework was tested, and 2) download performance of P2P systems, when integrated with PeerSelector, were tested.

We used all three metrics: score, RTT-based latency, and AS-Hop metrics to test the behavior of our framework. To evaluate profiling according to score metric, we tested the framework with a Unit test application. A code snippet is attached in Appendix B.6. 12 random peers were provided as input to PeerSelector for biasing according to score metric. Out of those 12, 9 peers were provided one by one using the *addpeer ()* API, and 3 peers were provided as a list of peers using the *addpeers()* API. The framework was successfully able to cluster those peers according to the score metric. The results at the console of the Unit test application were verified against PeerSelector's database.

Peer profiling according to other two metrics: RTT-based latency and AS-Hop metrics were also tested. In this experiment, a metric flag that was passed as a command line argument was changed to 1 and 2 for RTT-based latency and AS-Hop metrics respectively. To evaluate the history prioritization behavior, *getPeerImmediately()* API was called from the Unit test Application.

In the second phase of the evaluation, the framework was integrated with libtorrent library. The library and the *client_test* code were modified for this test. Objective of this test was to find the impact of peer sorting by PeerSelector on the deployed P2P system. Two different torrent files: one with regional users and another with global users, were used in the experiment. The experiment was conducted for all three metrics and was compared with the random metric. The metric type was passed as a command line argument to *client_test* client.

In order to test if the framework works for other P2P systems, it was also tested with Swift. The experiment was conducted in a manually controlled setup in PlanetLab.

5. Design

This chapter describes the design and implementation of the PeerSelector framework, the interface used between PeerSelector and transport protocols, as well as, the interaction of the framework with MySQL++ database.

The first section provides a design overview. The second section focuses more on framework's modules. The framework components are explained in further detail in this section. The third section focuses more on the design details and terminology used. Finally, the public interfaces are described in the final section.

5.1 Design overview

PeerSelector is an interest-based peer profiling framework. It is designed to group peers based on different policies. More specifically, in this thesis, those policies dictate that peers be grouped according to geographical locality, RTT-based latency, and nearby ASes. The framework design should be modular and therefore easily extendable. The design should allow for the addition of new policies in the framework, without major engineering modifications to the core engine. In addition, the framework should be easily integrated with P2P distributed protocols, for example, BitTorrent and Swift, but may also be integrated with other distributed protocols, such as peer discovery mechanisms. An overview of this design is illustrated in Figure 5.1.

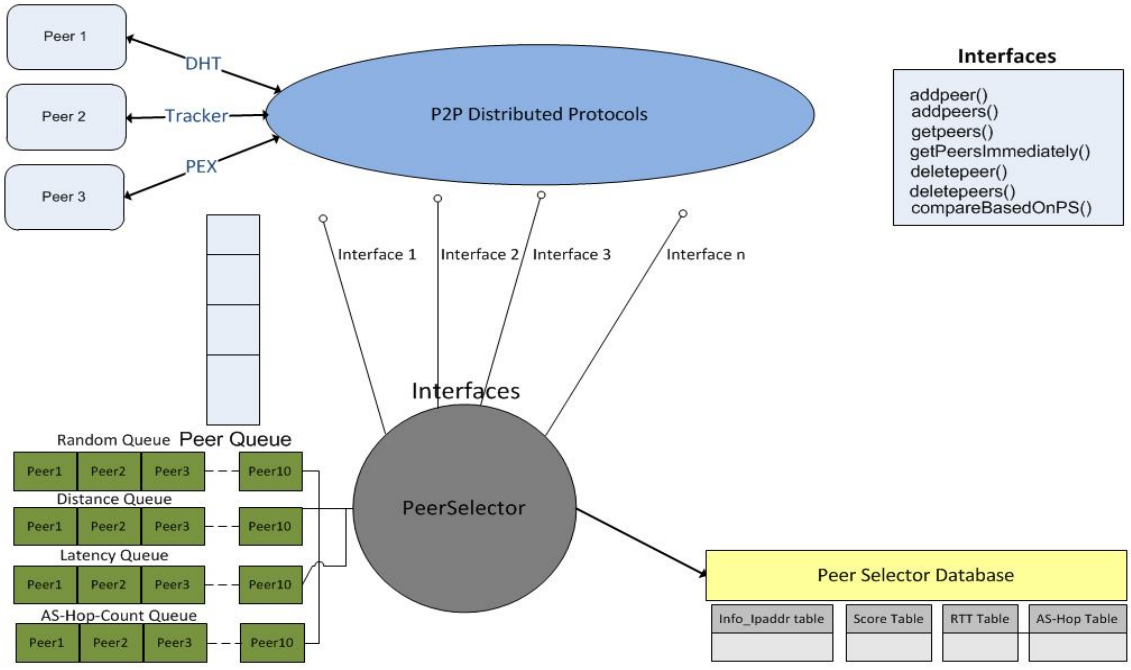

Figure 5.1 PeerSelector - Design

Firstly, PeerSelector groups peers based on certain policies. To group peers based on certain policies, PeerSelector uses the corresponding metrics to determine which peers are comparatively closer than other peers in a given neighbor set, exhibit lower latency, or are located in the nearby ASes.

To group peers according to geographic distance, PeerSelector uses score metric. The score metric uses geo-location to determine the distance between given peers. In order to distinguish local peers from remote peers, PeerSelector has to employ certain mechanisms that calculate a more accurate geographic distance. To do so, geo-location information like ASN, city, country and continent are included in the computation. This information is retrieved using publicly available geo-location APIs [34].

Moreover, PeerSelector can also group peers based on RTT-based latency metric. A peer with lower latency is preferred over a peer with higher latency. PeerSelector calculates latency in two different ways: using previous history or using proactive network probing.

In addition to the score and RTT metrics, PeerSelector employs another metric that groups peers according to nearby ASes --- the AS-Hop metric. The metric is used to find the number of AS-Hop that a peer needs to traverse to reach its neighbor.

Each peer selects its neighbor set based on its own interest. For example, if a peer wants to select a group of local peers, then it asks PeerSelector to group peers according to score metric. Similarly, if a peer wants to retrieve content from a given peer through the shortest AS path, it asks PeerSelector to group and provide him with the neighbor set that is closer in terms of AS-Hop counts.

PeerSelector stores the computed information about the peers in its database. When a request from user for peer grouping arrives the framework, it fetches those peers from the database and stores them in descending order in queues. Best peer is stored at the front of the queue, second best at second position from front, and so on. Queues are dynamic in nature and are refreshed whenever new preference interest is shown by the user. Current implementation supports four different types of queues. They are named as the random, distance, latency, and AS-Hop-Count queue as shown in Figure 5.1.

To communicate with different transport protocol, PeerSelector provides list of Interface APIs. So, interface is nothing but an APIs set exposed by the framework, which can be used by different P2P distributed protocols to notify the framework to add new set of peers, to give the new set of preferred peers, or to delete the unwanted peers. Current design provides interfaces for BitTorrent and Swift protocol however, they can be easily extended with minor modification and can be used with any other P2P systems.

5.2 PeerSelector's modules

PeerSelector application can be broadly divided into three major components: core, queues, and storage. In addition to this, engine exposes the interface for communication with Swift, BitTorrent and other P2P distributed systems.

5.2.1 Core

Core component consists of a class named BitSwiftSelector which in turn contains methods for achieving peer selector functionalities, attributes for storing data, threads for sorting peers based on certain policies, APIs for communicating with Swift and BitTorrent protocols and APIs for storing, retrieving, and deleting peers from the database. The APIs of the core module provides slots for adding additional policies and additional interfaces to distributed applications. This can be achieved by adding more peer preference functions --- which can prefer or cluster peers based on other criteria.

5.2.1.1 BitSwiftSelector

The BitSwiftSelector class is responsible for clustering peers based on certain policies. These policies are explained in section 5.2.1.3. Peer properties are defined using policies and are measured according to the metrics defined in section 5.2.1.2. See Appendix A.1 for a detailed class diagram. Figure 5.2 represents the core module.

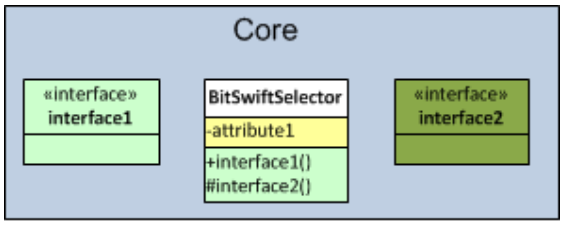

Figure 5.2 PeerSelector - Core component

5.2.1.2 Metrics

A metric is used to determine which peer is better compared to other peers**.** We use three different metrics to group peers. If the requirement is to group peers based on distance, then the score metric is used. If latency is the grouping criteria, then the RTT-based latency metric is used. Finally, if peers are to be grouped based on the AS-Hop count, then the AS-Hop metric is used.

These metrics are explained in detail as follows.

Score Metric - Our framework clusters peers based on geo-location using information like ASN, city, country and continent. We assign different weights to different information. For example, if peers are from the same ASN, then they are preferred over peers from different ASN. If peers are not from the same ASN, then we go one level up and compare peers based on city. So, every information is associated with some weight and it is added to the actual geographic distance between two peers. After adding the weighted value, we compute the score of a particular peer from another peer; the lower the score, the better the peer. This metric for calculating a better peer is useful in a scenario where the intention is to download/upload from local peers. Maxmind GeoIP database and GeoIP APIs are used for calculating the score [34].

RTT-based latency Metric - If the requirement is to download content from a peer which exhibits lower latency, then peer grouping is done based on the RTT-based latency metric. In

our design, this is done by probing a given peer three times in an interval of 200 ms and then calculating the mean RTT.

AS-Hop Metric - In order to cluster peers from nearby ASes, we use the AS-Hop metric. We calculate the shortest AS path to the peer. A peer with lower hop counts is preferred. The shortest path is represented in terms of hop count. The ASDistance database, generated using routing tables from the "RouteViews project", is used for calculating the distance between two IP addresses in terms of hop count.

5.2.1.3 Policies

Policies are nothing but a type according to which peers must be clustered. Each policy uses a respective metric to make its preference decision. Our current implementation of the framework supports four different types of policies. However, the design allows us to easily add as many policies as we want. In other words, we can treat each policy as one module and we can plug as many modules as required, with minor code changes in the core module. Four supported policies are named as random, less distance, lower latency, less AS-Hop count.

Random - As the name suggests, random policy clusters peers randomly. They are not biased towards any policies. The policy is very useful in evaluating the impact of our framework i.e. evaluating the performance of requesting protocol with and without locality biasing.

Less Distance - When geographically near peers are to be selected, then we apply the lower score policy. Peer preference decision is based on the score metric.

Low Latency - Low latency policy cluster peers according to the latency value. Latency can either be calculated using probing or can be used from the previously calculated and stored latency value from the database. In case of proactive probing, policy uses RTT-based latency metric to decide peers with lower latency.

Less Hop-Count - This policy aims to cluster peer from close by ASes. The policy can benefit ISPs by localizing the traffic within the same AS or within few nearby ASes. This can also benefit the user because after applying this policy, it is very likely that peers can download/upload content from/to close by ASes, which in turn, may improve the download/upload speed. This policy uses AS-Hop count as a metric, when making a preference decision.

5.2.2 Queues

The PeerSelector arranges peers in a container called queues; peers in one queue exhibit common properties. So, a queue contains preferred peers which are clustered according to certain preference metric. For example- nearby peers are preferred over far peers if queues are required to be populated according to score metric. Similarly, low latency peers get the priority, if queues are required to be generated based on RTT-based latency metric. In the first version of PeerSelector framework random, less distance, low latency, and less Hop-count policies are supported.

Queues are dynamic in nature and they are populated on demand. Integrated P2P distributed protocols make a choice which indicates according to what metric the peers should be clustered and kept in queues. Integrated protocol can also ask for the exact number of peers

exhibiting certain properties from PeerSelector. Based on the preference choice and the number of peers requested by the integrated protocol, queues can be refreshed with new set of peers.

Furthermore, to generate the queues, PeerSelector uses its storage. Since, queues are generated per policy, we support four different queues and they are generated as follows:

- After receiving the commands from the user of P2P distributed protocols, the core module contacts its storage component.
- It starts a search in database based on infohash/roothash and preference policy.
- Core then fetches peers according to Infohash/roothash, preference policy and the number of peers and stores them in the corresponding queues. If the numbers of peers are not specified then queues are populated with all the peers associated with any given infohash/roothash.
- Queues are populated per roothash/infohash. This means that, for two different files two different queues are generated.

Four different queues are shown in Figure 5.3.

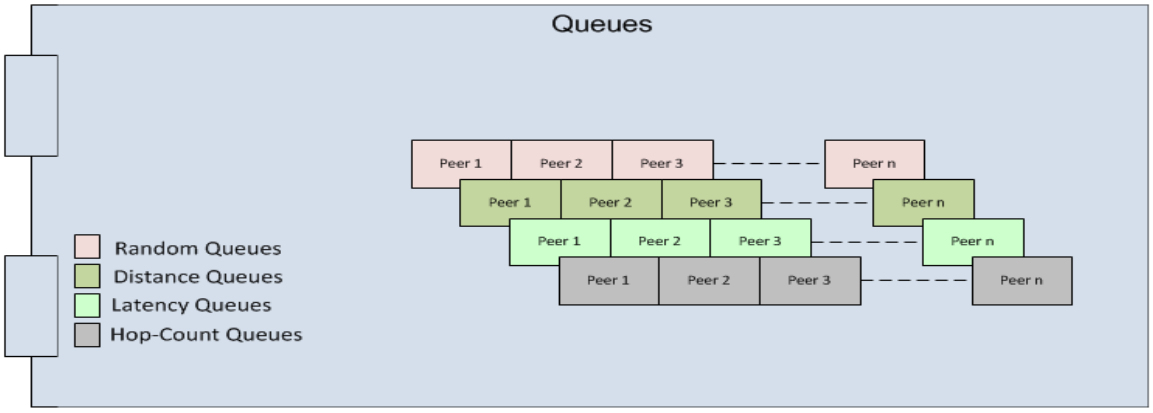

Figure 5.3 PeerSelector - Queues for storing preferred peers

5.2.3 Databases

In our design, we use the database to store information. The information is used for peer profiling. We use three major databases. First one, we call it the PeerSelector database or the main database. In addition to it, we use two external databases: the GeoIP database and the ASDistance database. Information about all the peers encountered over time is stored in the main database. The information can either be atomic information or derived information. Peer's IP address, port number and infohash/roothash are known as atomic information. Information which is computed from the atomic information is known as derived information, for example ASN, score, latency, AS-hop count and so forth.

The main database consists of several tables, where information about the encountered peers is stored. In our implementation we name those tables as: *infohash_ipaddr*, *score_table*, *rtt_table* and *ashop_table*. Detailed description of each table along with datatypes and primary keys are explained below.

infohash_ipaddr - The table is designed to contain peer's IP address, port number and infohash/roothash. Additionally, it contains the AS number which is computed by using the *getASN ()* API. The table uses a composite key consisting of: IP address, port and infohash/roothash as primary key.

score_table - The table is used to store score information. Score represents the geographic distance between two peers. Geo IP database is used for calculating the score of peers based on geo-location. Peers with lower score are considered better peers than peers with higher scores. The composite key consists of the IP address and infohash/roothash acts as a primary key. So, the combination of IP address and infohash/roothash should be unique across the table.

rtt_table - The table contains the IP address, port, infohash/roothash and latency information. Latency is calculated by probing the peer 3 times and then calculating the average latency after considering standard deviation. We update the RTT value only, if the recently calculated RTT is 50 ms different from the historical value. This design is made to reduce the number of frequent updates in the rtt_table. IP address and infohash/roothash act as the primary key and there should be no duplicate entries for the same set of IP address and infohash/roothash.

ashop_table - The purpose of this table is to store AS hop-count information. Table has four fields: IP address, port, infohash/roothash and the hop-count. Hop count is the distance between two IP addresses in terms of AS-Hop. It is calculated by using ASDistance database. IP address and infohash/roothash acts as a primary key and they must be unique across the table.

Moreover, storage also consists of two more databases: GeoIP database and ASDistance database. GeoIP database stores the location information. More specifically, it contains information about ASN, city, country and continent. Additionally, it contains longitude and latitude position which is used by PeerSelector to calculate the actual geographic distance between any two IP addresses.

Furthermore, the ASDistance database gives information about distance in terms of AS-Hops. The database is a JAVA serialized object and it contains BGP data. The database stores the AS number based on an index. AS hop count calculation is done using the *getDistance ()* method. The *getDistance ()* method returns the AS hop-count between two IP addresses. Steps are as follows:

- Two IP addresses are passed as arguments to the *getASN ()* method and a corresponding AS number is returned.
- Next, the index of the AS number in the ASDistance database is searched.
- Difference between two indices is the Hop count between two AS numbers.

PeerSelector makes use of the above two databases to populate the tables in the main database. The stored information is used in finding out peer properties. In the future, this process can be taken a step further where various queries can be run to find out interesting patterns for example - what are the most commonly used ASNs, and which ASNs always give the peers with lower latencies.

All three databases with various tables are shown in Figure 5.4.

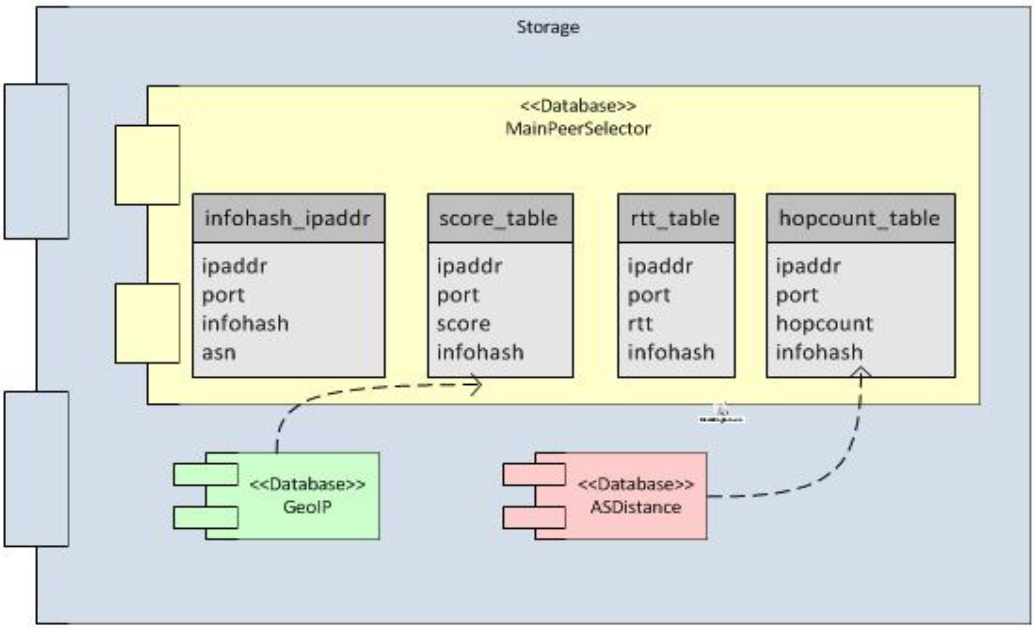

Figure 5.4 PeerSelector - Storage Component

5.3 High level design

5.3.1 Components

A PeerSelector component diagram is shown in Figure 5.5. After defining components in the previous section it was required to wire those components, and visualize the framework as a whole. Based on the components, framework was organized into classes. As indicated in the illustration, core component is responsible for interacting with all other components. It was required to define the behavior of core component in detail. Other components were external components and their behavior was already defined in previous sections. The class used in core module is named as BitSwiftSelector. The rest of classes are external to the framework and can interact with it using the interface exposed for communication. Since the initial evaluation of PeerSelector was carried out with BitTorrent and Swift protocols, a hybrid name for the main class was chosen. The behavior of the class is defined by its methods --- public, protected and private. External classes can use the functionalities provided by BitSwiftSelector class, using its public methods. Public methods can be invoked by other classes, by instantiating class objects. Class diagram is shown in Appendix A.1.

As shown in Figure 5.5 BitSwiftSelector class uses the database to store peer information. This information is retrieved and stored in queues when asked by P2P distributed protocols. Swift and BitTorrent are external component and users using these components ask PeerSelector framework to group peers.

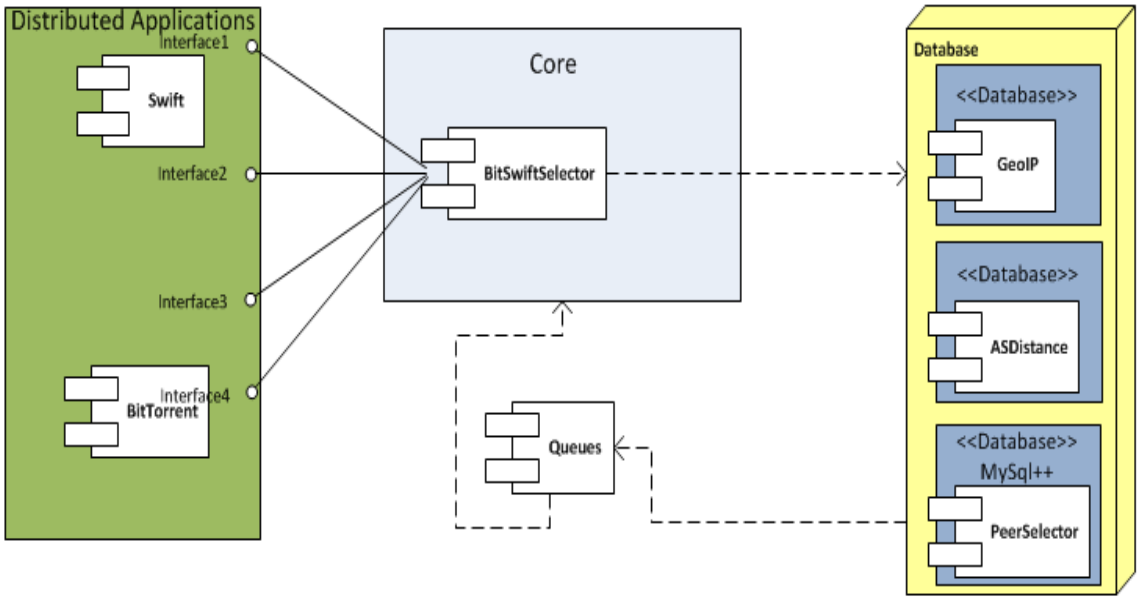

Figure 5.5 PeerSelector - Components Diagram

5.3.2 Workflow

This section covers the workflow of the messages, action between the components of the system and the sequence of interaction. UML provides an easy way to do this using sequence diagram. Below section depicts the different objects and the sequence of message exchanged between them to achieve the design goal.

P2P distributed protocols interact with PeerSelector framework, using interface exposed by the framework. Interface offers different handlers which are used by P2P distributed protocol to ask the framework to perform certain action according to user's indicated interest. These handlers are mainly *addpeer/addpeers, getpeers, getPeerImmediately, deletepeer/deletepeers* and *comapreBasedOnPS*. Flows of messages during call of these handlers are explained in subsequent sections.

5.3.2.1 addpeers

Figure 5.6 represents the flow of message of *addpeer/addpeers* API call for supplying peers to PeerSelector.

Steps for adding peers are explained as follows.

- When list of new peers associated with a given content are discovered by any distributed transfer protocol, newly learned peer's address, its port number, and infohash/roothash are relayed to PeerSelector for preference processing. This is done using *addpeer/addpeers* interface provide by PeerSelector.

- PeerSelector creates the separate non-blocking thread and the control is returned back to the distributed P2P protocol.
- The thread invokes *addPeerInDB()/addPeersInDB()* method of PeerSelector, which does the preference calculation and stores peer in PeerSelector database.
- *addPeerInDB()/addPeersInDB()* method first gets the AS number based on the IP address passed by the distributed protocol.
- It then creates the infohash_ipaddr table, where it stores all the IP addresses, port numbers and infohashes/roothashes along with AS numbers.
- PeerSelector then calls the *sortPeers()* method to arrange the peers based on the policies.
- *sortPeers()* method fetches the IP addresses and port numbers associated with the particular infohash/roothash from the infohash_ipaddr table.
- Now the preference calculation is done for these IP addresses, using calculation functions *calculateScore, calculateRTT* and *calculateHopCount* for score, latency and AS-Hop policies respectively.
- Finally, a list of preferred peers with preference metric is stored in descending order in the respective database's table.

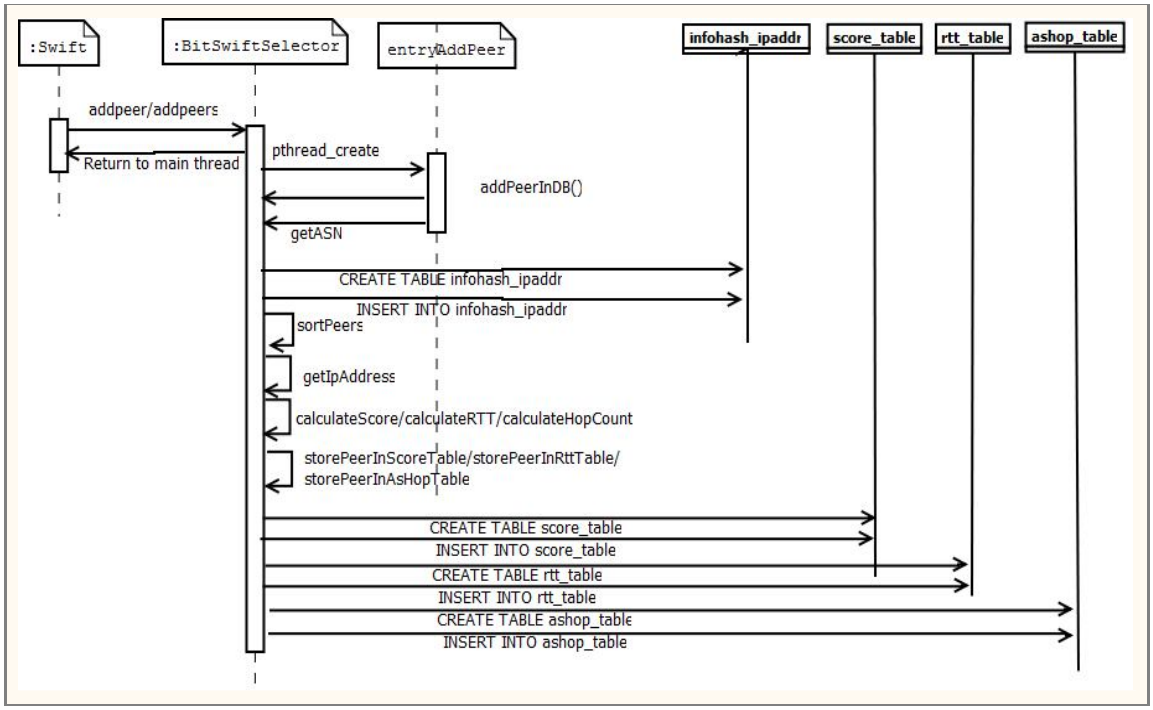

Figure 5.6 Sequence Diagram - addpeer/addpeers

5.3.2.2 getpeers

If the distributed protocol wants to request peers for a given infohash/roothash, it uses the *getpeers* API to contact PeerSelector. Peer preference calculation is done at run time. After

the calculation, a list of peers --- available in the respective queues --- is returned to the callee. Figure 5.7 depicts the sequence diagram for getting peers from the PeerSelector.

The steps for fetching peers from PeerSelector are described as follows:

- On a request for peers from a distributed protocol, PeerSelector starts the peer preference calculation for the policies specified in the parameter. In addition to the policy type, infohash/roothash and number of requested peers are also passed as parameters to PeerSelector. These parameters are passed as an argument by the P2P distributed protocol users.
- PeerSelector then calls the *getIpAddress* method to get the list of peers associated to the passed infohash/roothash. The method tries to retrieve IP addresses and port numbers from the infohash_ipaddr table in the database.
- After retrieving the list of IP addresses and port numbers, PeerSelector starts the preference calculation for the policy type specified in the method call. If the policy type is 0 then *calculateScore* method is triggered, if the type is 1 then *calculateRTT* method is triggered, and if the supplied policy type is 2 then *calculateHopCount* method is triggered.
- Once the calculation is completed, peer set is arranged in descending order based on the specified metric and is returned to the callee. As the set is calculated and generated at run time, the getpeers call may take longer time for execution.

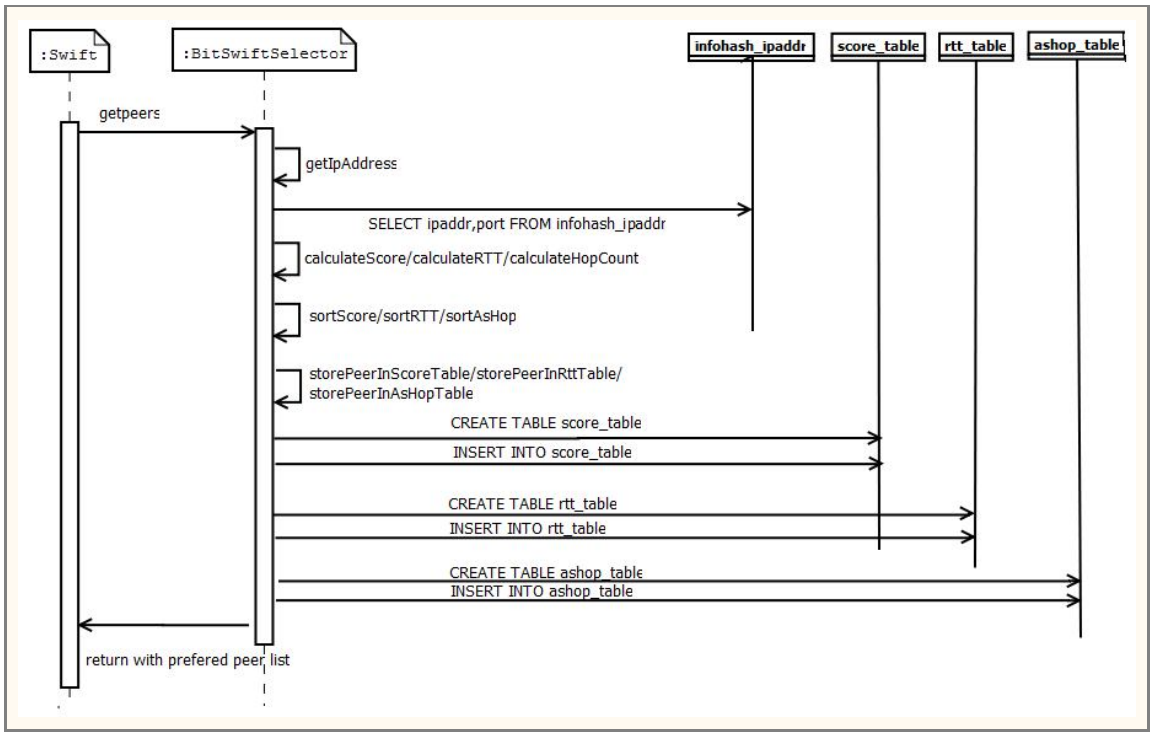

Figure 5.7 Sequence Diagram – getpeers

5.3.2.3 deletepeers

When distributed protocols no longer find certain peers --- associated with a given infohash --- useful, they may ask for alternative peers instead. In such scenario, the distributed protocol requests PeerSelector to delete all peers related to a given file first. Message interaction in sequential order is shown in Appendix A.2.

Steps required to delete the set of peers associated with a given file are explained below.

- First, *deletepeers* API from any distributed protocol is invoked. Infohash/roothash of the file, whose peers are no longer useful, and option policy type are passed as an argument.
- If the policy type is passed as an argument, then peers from the database table associated with the specified type are deleted. If not, then all the peers related to the given infohash/roothash across all tables are deleted.

5.3.2.4 deletepeer

If a distributed protocol does not want to use certain peer as it has become non-responsive, then it asks PeerSelector to delete that peer. The protocol in question can instruct PeerSelector to delete peers from any specific table or from all tables. In addition, the protocol can ask for a new set of peers using the *getpeer* or *getPeersImmediately* APIs. *getPeersImmediately* API is explained in next section. The sequence of interactions needed to carry out the *deletepeer* process is illustrated in Appendix A.3.

Steps for one peer deletion shown in Appendix A.3 are described as follows. All the steps are similar to the steps mentioned in 5.3.2.3 section except for the input parameter.

- When a particular peer is no longer useful by a distributed application, then it informs this to PeerSelector using *deletepeer* API call.
- Upon deletion request, PeerSelector checks the policy type.
- If the policy type is specified, then PeerSelector searches for peers in the respective table. Peer search is done using the IP address, port number and infohash/roothash.
- If the peer is found, then it is deleted from the corresponding table.
- If the type is not mentioned by the distributed application, then the peer is searched across all tables in the database and all peer occurrences are deleted.

5.3.2.4 getPeersImmediately

When a distributed application wants to get the list of peers urgently --- without waiting for PeerSelector to compute peer preferences --- it uses *getPeersImmediately*. This API is used to populate the queue urgently. The corresponding sequence diagram is available in Appendix A.4.

The steps to request the already stored peers from the database are explained below:

- A distributed application requests peers without any delay.
- On request, PeerSelector checks the type and infohash/roothash value.
- PeerSelector then contacts the database and tries to fetch the preferred peers from the table associated with the policy type passed as an argument.
- Once the list of peers is ready, PeerSelector sends it back to the distributed application.

5.3.2.5 compareBasedOnPS

PeerSelector provides support for comparing two peers based on certain policies. In our current implementation, this API is called by the libtorrent implementation at the time of selecting candidates for downloading. Messages exchanged can be viewed in the sequence diagram in Appendix A.5.

Peer comparison steps are carried out in the following way:

- To accomplish this task, PeerSelector contacts the database twice. First it gets the metric value related to the first peer.
- Then it contacts the database again and fetches the metric value for the second peer.
- It then compares the two values and return true if first peer is better than second peer otherwise it returns false.

5.4 Public APIs

List of interface APIs and their description is presented in Table 5.1.

Name	Description
addpeer	This API is used by the P2P distributed protocol to notify PeerSelector about a newly discovered peer. After receiving the notification, the framework stores the peer in its storage. PeerSelector then sorts the list of already available peers --- along with the newly discovered peer --- and keeps it ready for use.
addpeers	This API is same as addpeer except the fact that instead of one peer, it takes list of peers as input.
deletepeer	This API is used by the transport protocol to notify PeerSelector to remove an unwanted peer. After receiving the notification, PeerSelector deletes that particular peer from the storage. The flexible design allows the transport protocol to specify the type of storage from which unwanted peer has to be removed. If the specified type is 0, then peer is removed from score table. If the type relayed by transport protocol is 1, then the peer is removed from the rtt table. If the as hop information is no longer useful, then the transport protocol

	supplies type as 2. If no type is specified, then the peer is removed across the main table and the rest of policy-specific tables.
deletepeers	This API is used by the transport protocol to notify PeerSelector to remove all unwanted peers related to specific infohash/roothash. After receiving the notification, PeerSelector deletes all the peers related to any given infohash/roothash.
getpeers	This API is used by the transport protocol to ask for the preferred peers for a given infohash/roothash from the framework. After receiving the request, PeerSelector starts the computation for preferred peers, depending on the indicated preference, and returns the list of preferred peers. The API provides the flexibility to the user to specify the number of peers. If the count is not specified then all the peers associated with any given info-hash are returned.
getPeerImmediately	This API is similar to the getpeers API except for the fact that it does not compute preferred peers based on preference but based on a previously computed value. The API is used by the transport protocol to ask for the preferred peers for a given infohash/roothash from PeerSelector. After receiving the request, PeerSelector fetches the list of preferred peers from the storage and returns them. This API is designed to reduce the calculation time. API provides the flexibility to the user to specify the number of peers. If the count is not specified then all the peers associated with any given info-hash are returned.
compareBasedOnPS	The API is used to compare peers based on the preference type. In the current design it is used by the libtorrent protocol to find the connection candidate. It compares two peers for a given infohash/roothash based on the indicated interest. True is returned if left peer is better than right peer otherwise false is returned.

Table 5.1 List of interface APIs

6. Implementation

This chapter shows the main contributions of this project. The chapter defines different methods used inside PeerSelector. Framework functionality is implemented in C++ programming language. The first subsection presents the public interface and code snippets to understand the external functionalities. Secondly, a list of protected methods and description about them are provided. The third section presents the mechanism to communicate with the database. Finally, the fourth section shows the modification made in BitTorrent protocol in order to integrate PeerSelector with it.

6.1 Interface

PeerSelector provides an interface to distributed applications described in Listing 6.1. Following interface handlers are used for communication by the integrated distributed protocols.

Listing 6.1 Implementation - Interface

```
class BitSwiftSelector{
public:
    BitSwiftSelector();
    ~BitSwiftSelector();

     // stores the notified list of peers in its storage
     void addpeer(string ip, int port, string hash);

    // delete the specified peer from the storage
     void deletepeer(string ip, int port, string hash, int type = 100);

    // delete all the peers associated with a particular file
     void deletepeers(string hash, int type = 100);

     // retrieve the list of peers based on policies measured by metrics
     void getpeer(string hash, int type, std::vector<ipPort_s> &ip_port, int count = 0);

     // get peers from the database without introducing any delay
     void getPeerImmediately(string hash, int type, std::vector<ipPort_s> &ip_port, int count =
0);
     // compare peers based on policy type
     bool compareBasedOnPS(string ip1, string ip2, string hash, int type);
          ...
};
```

6.2 Protected methods

Protected methods are internal to the framework and are responsible for providing core functionalities. Few important methods are explained along with the source code in the below section. List of protected methods can be found in Appendix F.1-F.19.

6.2.1 Finding longitude and latitude

PeerSelector requires the longitude and latitude information to calculate the geographic distance between two peers. This information is available from the MaxMind GeoIP database and can be retrieved using the GeoIP API. A code snippet is available in Listing 6.2.

Listing 6.2 Implementation - Getting Longitude and Latitude information

```
void BitSwiftSelector::getCoordinates(char *ip, double &latitude, double &longitude)
{
    try
    {
        GeoIPRecord    *gir = GeoIP_record_by_addr(m_gi, ip);
        if (gir)
        {
            latitude = gir->latitude;
            longitude = gir->longitude;
        }
    }
    catch (...)
    {
    }
}
```

6.2.2 Calculating score

After getting the longitude and latitude information, it was required to calculate the geographic distance between two peers. After calculating the geographic distance, a weight factor is added to it, in order to prefer ASN, city, country, and continent, in descending order. This weighted value is called a score. The calculation mechanism is shown in Listing 6.3. Score is calculated as follows:

- First, Longitude and latitude value of two IP addresses is obtained by using the *getCoordinate* method.
- Then, geographic distance between two points is calculated by using the *calculateDistance* method. Refer Listing B.7.

- After that, a weight factor is added to the distance value. ASN has priority over city, which has priority over country, and country is preferred over continent. Lower score is considered as better score. For example - consider two peers are at equidistance from requesting peer and one is from the same ASN however other is not from the same ASN but from the same city; then the peer from same ASN is considered as the better peer.

Listing 6.3 Implementation - Calculate score

```
int BitSwiftSelector::calculateScore(char* peerIp)
{
…
  // get longitude and latitude position
  getCoordinates(peerIp, peerLatitude, peerLongitude);
  getCoordinates(BitSwiftSelector::m_myIp,myLatitude,myLongitude);
  // calculate distance between two points
  double distance = calculateDistance(peerLatitude, peerLongitude, myLatitude, myLongitude);
   // assign weight to particular information and add weight to the actual distance to calculate weighted distance
   if (isInSameASN(peerIp))
   {
      extraFactor = 1;
   }
   // ISP database is not free
   /*else if (isInSameProvider(peerIp))
   {
      extraFactor = 10;
   }*/
   else if (isInSameCity(peerIp))
   {
      extraFactor = 100;
   }
   else if (isInSameCountry(peerIp))
   {
      extraFactor = 200;
   }
   else if (isInSameContinent(peerIp))
   {
      extraFactor = 300;
   }
   else
   {
      extraFactor = 500;
   }
   totalScore = distance + extraFactor;
   return totalScore;
}
```

6.2.3 Calculating RTT

In order to calculate RTT, peer's IP address is probed 3 times. Approach mentioned in "A Top-down Approach Featuring the Internet" book is used for calculating the EstimatedRTT and timeout [37]. However, the current implementation relies on the ping protocol for timeout and the Retransmission Timeout (RTO) value can be used in the future. Refer Listing 6.4 and Listing 6.5.

Listing 6.4 Implementation – Calculate RTT.sh

```
#!/bin/sh

TTL_STRING="$(ping $1 -i .2 -c 1 | grep 'rtt' | awk '{ print $4}')"

TTL=$(echo $TTL_STRING | cut -d '/' -f2)

echo $TTL
```

Listing 6.5 Implementation - Calculate RTT

```
double BitSwiftSelector::calculateRtt(char* ipAddress)
{

    FILE *fp;
    char rttBuffer[20];
    double SampleRTT = 0;
    double EstimatedRTT = 0;
    double Deviation = 0;
    double RTO = -1;
    double x = .1;
    for (int i = 0; i < 3; i++)
    {
        // call rrt.sh for calculating rtt by probing the ipAddress
        char cmd[50] = "/bin/sh rtt.sh ";
        strcat(cmd, ipAddress);

        fp = popen(cmd, "r");

        if (fp == NULL)
        {
            cout << "Failed to run command"<< endl;
            exit(1);
        }
```

```
        // Read the output one line at a time
        fgets(rttBuffer, sizeof(rttBuffer)-1, fp);
        SampleRTT = atof(rttBuffer);
        if (SampleRTT == 0)
        {
            break;
        }
        if (RTO == -1)
        {
           EstimatedRTT = SampleRTT;
           Deviation = SampleRTT / 2.0;
           RTO = EstimatedRTT + 4 * Deviation;
        }
        else
        {
          // since the IP address is unreachable 2nd time, so there is no point trying again
            if (EstimatedRTT == 0)
               break;
            Deviation = (1 - x) * Deviation + x * abs(EstimatedRTT - SampleRTT);
            EstimatedRTT = (1 - x) * EstimatedRTT + x * SampleRTT;
            // This time out is for future use, right now I am relying on ping protocol for timeout
            RTO = EstimatedRTT + 4 * Deviation;
        }
        pclose(fp);
    }
    return EstimatedRTT;
}
```

6.2.4 Calculating AS-Hop

PeerSelector supports clustering of peers according to the AS hop count. Hop count is calculated using following steps. Code snippet is available in Listing 6.6, 6.7, and 6.8.

- Inside *calculateHopCount* method of PeerSelector, *as-hop.sh* script is called. Then IP addresses of the peers between which the hop count has to be calculated are passed as an argument.
- Inside *as-hop.sh* script, classpath for asdistance package is set, and then java executable *ASHop*, with the IP addresses of the peers, is executed.
- Inside ASHop.java file, ASDistance package [30] is imported and ASDistance class is instantiated. After that, *getDistance* method is called to get the AS-Hop count between two IP addresses.

Listing 6.6 Implementation - Calculate AS-Hop count

```
int BitSwiftSelector::calculateHopCount(char* ipAddress1, char* ipAddress2)
{

    FILE *fp;
    char asHopBuffer[10];
    if (ipAddress1 == 0 || ipAddress2 == 0)
    {
        cout << "ERROR : IP address is NULL"<< endl;
        return -1;
    }
    char cmd[50] = "/bin/sh as-hop.sh ";
    strcat(cmd, ipAddress1);
    strcat(cmd, " ");
    strcat(cmd, ipAddress2);

    // Pass source and destination IP addresses
    fp = popen(cmd, "r");

    if (fp == NULL)
    {
        cout << "ERROR : Failed to run command"<< endl;
        return -1;
    }

    // Read the output, one line at a time
    fgets(asHopBuffer, sizeof(asHopBuffer)-1, fp);
    pclose(fp);
    return atoi(asHopBuffer);
}
```

Listing 6.7 Implementation - ASHop.sh

```
#!/bin/sh
PACKAGE_PATH=`pwd 2>&1`
export CLASSPATH=$PACKAGE_PATH/as-distances-1.0/se/sics/asdistances:.
cd $PACKAGE_PATH/as-distances-1.0
AS_HOP="$(java se.sics.asdistances.ASHop $1 $2)"
echo $AS_HOP
```

Listing 6.8 Implementation - ASHop.java

```
package se.sics.asdistances;
import java.io.*;
import java.util.Map;
import java.util.logging.Level;
```

```
import java.util.logging.Logger;
import java.util.zip.*;

public class ASHop{

   public static void main(String args[])
   {
      ASDistances distances = ASDistances.getInstance();

      // Error check for number of arguments
      if(args == null || args.length == 0)
      {
         System.out.println("Pass two IP addresses as argument");
         System.exit(0);
      }

      // Get AS Hop count between two IP addresses
      byte d = distances.getDistance(args[0], args[1]);
      System.out.println(d);
   }
}
```

6.3 Communication with the database

PeerSelector often communicates with the database for storing or retrieving peer information. In order to communicate with the database, PeerSelector has to open a connection with the database server. It then executes different queries. If the query does not return any result, like a CREATE query, then only pass and failed status are stored in a special result type called a *SimpleResult*, otherwise results are stored in a result object of *StoreQueryResult* class. Any errors are handled by the exceptions thrown by the library and can be printed using the query.error() message. A code snippet in Listing 6.9 describes communication procedure.

MySQL++ APIs used for database communication and their usage patterns are listed below:

- First a connection object was created to communicate with database server.
- Then a query object was created using the connection object. The query object was from query class.
- A connection was established with the database server by providing host name, user name, and the password. *connect* API is used for creating connection.
- Once the connection was established with the database server, then the database was created, if it was not present. *create_db* and *select_db* APIs of connection class were used for this purpose. Name of the database used in this project is *peerselector*.
- Next step was to create SQL table, where peer information is saved. *CREATE TABLE* query was executed if the table was not present.

- Once the table was created, different operations on table were performed like: inserting data, selecting data or deleting data. To insert data into the table, *INSERT INTO* query command was executed.
- Another operation on SQL table which was important in our scenario was to retrieve the already stored peer information. Depending upon the policy type, *SELECT* query was run on different tables to get the information about any peer. Those tables are: *score_table, rtt_table* and *ashop_table.*
- Then a result object was created to store the information returned by the *SELECT* query. The object was from *StoreQueryResult* class.
- The result object --- res was iterated till the end to get the data from each row.
- MySQL++ also provides a convenient way to delete data from the table. In order to perform delete operations on the table, *DELETE* query was executed.

Listing 6.9 Implementation - Communication with database

```
#define DB "peerselector"                 // name of database
#define HOST "localhost"                  // location where sql server is running
#define USERNAME "root"                   // user of the database
#define PASSWORD "***********"      // Password for the above user
……..
…….
   // Created connection object
   mysqlpp::Connection conn(false);

   // Get an object from Querry class
   mysqlpp::Query query = conn.query();

   // Try to connect
   if (conn.connect("", HOST, USERNAME, PASSWORD))
   {
     if(!conn.select_db(DB))
     {
         conn.create_db(DB); // create database if database is not present
         conn.select_db(DB); // select database
      }

      // Check if the table is present
      mysqlpp::Query ifTableExist = conn.query("describe infohash_ipaddr");
       if (!ifTableExist.execute())
       {
          // create infohash_ipaddr table
          query << "CREATE TABLE infohash_ipaddr (ipaddr VARCHAR(156) not null , port
          INT not null , infohash VARCHAR(40) not null, asn INT not null, PRIMARY KEY
          (ipaddr, infohash))";
          if(query.execute()) // execute it!
          {
                 query.reset();

                  // insert peer information into infohash_ipaddr table
                 query << "INSERT INTO infohash_ipaddr(ipaddr, port, infohash, asn) VALUES
                 (\"" << ip << "\", \"" << port << "\" , \"" << hash << "\" , \"" << asn << "\")";
```

```
                query.execute();
            }
        }

        if (criteria == 0)
        {
                // Retrieve score information of a particular info hash from score table
                query << "SELECT ipaddr,port FROM score_table WHERE infohash = \"" <<
                file_hash << "\" ORDER BY score";
        }
……..
……..
        // Result object stores the result
        if (mysqlpp::StoreQueryResult res = query.store())
        {
                mysqlpp::StoreQueryResult::const_iterator it;

                // iterate till the end of res
                for (it = res.begin(); it != res.end();++it)
                {
                        mysqlpp::Row row = *it;
                        addr.ipAddress = const_cast<char *>(row["ipaddr"].c_str());
                        addr.port = row["port"];
                        ……..
                        ……..
                }
          }
          else
          {
                cerr << "Failed to get item list: " << query.error() << endl;
          }

          // delete peer from score_table
          if(criteria == 0)
          {
                query.reset();
                query << "DELETE from score_table where ipaddr = (\""<<ip<<"\") AND port =
                (\""<<port<<"\") AND infohash = (\""<<hash<<"\")";
                 if(!query.execute())
                        cerr << "Error : peer is not present in table: " << query.error() << endl;
          }
    }
    return;
}
```

6.4 Modifications in libtorrent

The main P2P distributed protocol used for this project is BitTorrent. In order to bias the peer selection mechanism of BitTorrent, modifications were needed both at the BitTorrent client side and inside the BitTorrent library. As mentioned earlier, the library used for the purpose of project is libtorrent-rasterbar-15.0.9.

6.4.1 Modifications in libtorrent client

Libtorrent client code was modified in order to implement peer biasing functionalities inside BitTorrent.

The following modifications were needed in *client_test.cpp* file. A code snippet is found in Listing 6.10.

- An object of BitSwiftSelector class was created inside main () function of *client_test.cpp* file. Object was used for calling different public methods of BitSwiftSelector class.
- Name of the torrent file was passed as a first argument to *client_test*. Refer section 7.2.3.1 for more detail.
- The infohash associated with torrent file was obtained.
- List of all the peers supplied by the tracker was obtained using the get_peer_info method [13].
- After obtaining the list of peers, these peers were added to the database, using *addpeers* API of PeerSelector.
- Once the peers were added to the database, *getpeers* or *getPeersImmediately* API of PeerSelector was used to get the list of sorted peers. *getPeersImmediately* API was used to get the peers immediately, without spending calculation time. On the other hand, *getpeers* API was used to obtain the list of peers, after calculating the peers' information available in database. After this step, a sorted list of peers was returned to client. These peers were fed back to the BitTorrent application in order to start the download with the sorted list of peers. In order to so, the download was paused and was resumed with new set of peers.
- *sort_peer_based_on_peerselector* API was called to change the *m_peers* list inside the *pollicy.cpp* file. The API was a newly added and a detailed description can be found in the subsequent section.

Listing 6.10 Implementation - Modifications in client_test.cpp file

```
/******************Modified to integrate PeerSelector functionalities ************************/

// Instantiate BitSwiftSelector class
BitSwiftSelector *objSelector = new BitSwiftSelector();
torrent_handle h = get_active_torrent(handles);
if (h.is_valid()){
std::vector<peer_list_entry> peers;
std::vector<addr_list> addrVector;
addr_list addr;
ipPort_s listIpPort;
std::vector<ipPort_s> ip_port_list, ip_port_list_add;

// Pass .torrent file as first argument
int size = file_size(argv[1]);
if (size > 10 * 1000000)
```

```
{
        std::cerr << "file too big (" << size << "), aborting\n";
        return 1;
}
std::vector<char> buf(size);
std::ifstream(argv[1], std::ios_base::binary).read(&buf[0], size);
lazy_entry e;
int ret = lazy_bdecode(&buf[0], &buf[0] + buf.size(), e);

if (ret != 0)
{
        std::cerr << "invalid bencoding: " << ret << std::endl;
        return 1;
}

torrent_info t(e, ec);
if (ec)
{
        std::cout << ec.message() << std::endl;
        return 1;
}

// get the infohash associated with .torrent file
char ih[41];
to_hex((char const*)&t.info_hash()[0], 20, ih);

// get list of peers supplied by tracker
h.get_peer_info(peers);

for (std::vector<peer_info>::const_iterator i = peers.begin(); i != peers.end(); ++i)
{
       // check for ipv6 address
        if (!(i->ip.address().is_v6()))
        {
                listIpPort.ipAddress = i->ip.address().to_string(ec).c_str();
                listIpPort.port = i->ip.port();
                ip_port_list_add.push_back(listIpPort);
        }
  }

// add list of peers to database
objSelector->addpeers(ip_port_list_add, ih);

// Second argument is the policy type
int type = 0;

//0/1/2 score/rtt/as-hop
if (!strcmp(argv[2],"0"))
        type = 0;
else if (!strcmp(argv[2],"1"))
        type = 1;
else if (!strcmp(argv[2],"2"))
        type = 2;
else if (!strcmp(argv[2],"3"))
```

```
        type = 3;

// Third argument is used for generating the view urgently or non urgently. If argv[3]  then
//generate non-urgent view and if argv[3] is 1 then generate urgent view.
if (!strcmp(argv[3],"0"))
        objSelector->getpeers(ih, type, ip_port_list);
if (!strcmp(argv[3],"1"))
        objSelector->getPeersImmediately(ih, type, ip_port_list);

// Sorted list of peers are ready for use. Store it in a vector
for (std::vector<ipPort_s>::const_iterator j = ip_port_list.begin(); j != ip_port_list.end(); ++j)
{
        addr.ipaddress = j->ipAddress;
        addrVector.push_back(addr);
}

// Pause the download and resume the download with new set of peers which are sorted
//based on policy
std::cout << "Pausing" <<std::endl;
h.pause();
if (type != 3)
{
   // Sort the m_peers list inside policy.cpp class and establish a connection with peers
   // from this list
   h.sort_peer_based_on_peerselector(addrVector, type);
}
std::cout << "Resuming" <<std::endl;

// Resume the download with sorted peers
h.resume();

}// end of h.is_valid()
/*************************************Modification Ends*********************************/
```

6.4.2 Modifications in libtorrent library

The libtorrent library also has to be modified, to integrate PeerSelector functionalities inside it. The following methods are added to the existing libtorrent library code. Source code can be found at GitHub repository [36]. Code snippets can be found in Listing 6.11, 6.12 and 6.13.

sort_peer_based_on_peerselector

- As mentioned in section 6.4.1, the client code fetches the list of peers --- arranged according to PeerSelector. This list is passed as an argument to the *sort_peer_based_on_peerselector* method for sorting. The sorted list is saved inside the policy class using the *storeAddr* method. User's indicated interest is also saved in a member variable inside *storeAddr* method.
- m_reset_round_robin, and m_peer_selector flags are added inside the policy class to enable peer biasing functionalities.

Name	sort_peer_based_on_peerselector
Input Parameter	std::vector<addr_list> addr - List of peers obtained from PeerSelector int type - Policy Type
Output Parameter	Void
Description	The API is added inside Torrent.cpp file to enable PeerSelector functionalities inside libtorrent library. To add the definition of API inside Torrent.cpp file, it was also required to add the API inside Torrent_handle.cpp file.

Table 6.1 Modified libtorrent method - Enable PeerSelector functionalities inside libtorrent

Listing 6.11 Implementation - sort_peer_based_on_peerselector

```
void torrent::sort_peer_based_on_peerselector(std::vector<addr_list> addr, int type)
{
        error_code ec;
         // store list of peer's addresses inside policy class
         m_policy.storeAddr(addr, type);

        // Enable Peer Selector functionalities by enabling flags related to Peer selector, which
        //are inside policy class
         m_policy.decide_based_on_peerSelector(true);
         m_policy.set_round_robin(true);
}
```

rearrange_peers

- As the name suggests, the rearrange_peers method arranges the peers' list obtained from tracker, according to PeerSelector. The peers given by tracker are placed in the m_peers data structure.
- The peers inside the m_peers data structure are compared against the peers' list obtained from the PeerSelector.
- After every swap operation, the better peer is pushed upwards in the list and eventually the entire list is arranged according to the list supplied by the PeerSelector.

Name	rearrange_peers
Input Parameter	std::vector<addr_list> addr - List of peers obtained from PeerSelector
Output Parameter	void
Description	This method is added inside policy.cpp file to rearrange the list of peers according to the indicated interest shown by users.

Table 6.2 Modified libtorrent method - Rearrange peers according to Preferred Interest

Listing 6.12 Implementation - rearrange_peers

```
void policy::rearrange_peers(std::vector<addr_list> addr)
{
        error_code ec;
        bool found = false;
        int k = 0;
        int count  = 1;
        for (int i =0; i < addr.size(); i++)
        {
                if (i == 0)
                    k = i;
                // if a peer is not found then swap index should not be incremented
                if (i != 0 && found != true)
                {
                        k = i-count;
                        count++;
                }
                 for (int j = k; j < m_peers.size(); j++)
                 {
                        // if a peer is not found in m_peers list then continue and compare the
                        // next peer
                        if (addr[i].ipaddress != m_peers[j]->ip().address().to_string(ec).c_str())
                        {
                                found = false;
                                continue;
                        }
                        else
                        {
                                // swap the peers and place better peer before other peers
                                 swap(m_peers[k], m_peers[j]);
                                k++;
                                found = true;
                                break;
                        }
                 }
        }
        return;
}
```

compare_peer_for_peerSelector

This method is added to find a better candidate for initiating the connection. Although the current libtorrent library has support for peer biasing, it was not sufficient to capture all the policies mentioned in this project. Existing peer biasing mechanism tries to find out the distance of left peer and right peer from the fixed IP address and a lower distance peer is selected for connection.

To support all three policies mentioned above and to bias peer selection more accurately, a new method is added inside the policy class which can compare two peers according to the PeerSelector's selection criteria.

Name	compare_peer_for_peerSelector
Input Parameter	policy::peer const& lhs - Left peer policy::peer const& rhs - Right peer
Output Parameter	bool - True if lhs a better connect candidate than rhs
Description	This method is added inside policy.cpp file to prefer better peer. Better peers are decided according to failcount, local peers and then PeerSelector.

Table 6.3 Modified libtorrent method- Decide better peer for connection

Listing 6.13 Implementation - compare_peer_for_peerSelector

```
// this returns true if lhs is a better connect candidate than rhs
bool policy::compare_peer_for_peerSelector(policy::peer const& lhs, policy::peer const& rhs)
const
{
   // prefer peers with lower failcount
   if (lhs.failcount != rhs.failcount)
   {
      // log peer information with failcount value
     fprintf(fp2, "Failcount %s   %s   %d        %d | ", lhs.address().to_string().c_str(),
                                      rhs.address().to_string().c_str(), lhs.failcount, rhs.failcount);
     return lhs.failcount < rhs.failcount;
   }
   // Local peers should always be tried first
   bool lhs_local = is_local(lhs.address());
   bool rhs_local = is_local(rhs.address());
   if (lhs_local != rhs_local)
   {
      // log peer information with local flag
     fprintf(fp2, "local %s   %s   %d   %d | ",lhs.address().to_string().c_str(),
                                       rhs.address().to_string().c_str(), lhs_local, rhs_local);
     return lhs_local > rhs_local;
   }

   // get the infohash associated with .torrent file
   char ih[41];
   to_hex((char const*)&m_torrent->torrent_file().info_hash()[0], 20, ih);
```

```
    // Pass IP address of the two peers long with policy type
    bool lhs_better = m_objSelector->compareBasedOnPS(lhs.address().to_string().c_str(),
                                                      rhs.address().to_string().c_str(), ih, m_type);
    // log the peer information according to the decision made by compareBasedOnPS
    fprintf(fp2, "PS   %s%s       %d | ", lhs.address().to_string().c_str(),
                                                      rhs.address().to_string().c_str(), lhs_better);
    return lhs_better;
}
```

In addition, the following existing methods are also modified. The methods are mentioned in Listing 6.14 and Listing 6.15. A small code modification is done in the *add_peer* method of policy class. The method is modified to support the add peer feature of the framework inside the library. Modified implementation allows the *add_peer* method to add peers in the PeerSelector database. Code changes are shown in Listing 6.14

Listing 6.14 Implementation - add_peer

```
policy::peer* policy::add_peer(tcp::endpoint const& remote, peer_id const& pid
            , int src, char flags)
{
…
...
    if (m_peer_selector != false && m_reset_round_robin == false)
    {
        char ih[41];
        error_code ec;
        to_hex((char const*)&m_torrent->torrent_file().info_hash()[0], 20, ih);

        // reset flag to start from the beginning in the sorted peers list
        m_reset_round_robin = true;
      m_objSelector->addpeer((*iter)->ip().address().to_string(ec).c_str(), (*iter)->ip().port(),
ih);
     }
…
…
}
```

Furthermore, to implement the peer biasing functionalities inside libtorrent, the *find_connect_candidate* method shown in Listing 6.15 is modified. This method finds the right candidate for establishing the peer-to-peer connection. This method is changed extensively from the functionality point of view.

- In order to start the candidate search from beginning inside m_peers list, m_round_robin flag is reset to 0.
- Then the current candidate is compared with the one which m_round_robin points to. The current candidate is passed as the first argument to the

compare_peer_for_peerSelector method and the peer which m_round_robin points to, is passed as the second argument.

- If m_peer_selector flag is false and m_type is 3, then existing libtorrent implementation for peer selection is used.
- Log file is saved with the name PreferredList, which contains the sorted list and the connection candidates.

Listing 6.15 Implementation - find_connect_candidate

```
policy::iterator policy::find_connect_candidate(int session_time)
{
…
...
   // reset m_round_robin to start from the beginnng in the sorted peers list
   if (m_peer_selector != false && m_reset_round_robin != false)
   {
      m_round_robin = 0;
      m_reset_round_robin = false;

      // Sort the list
       rearrange_peers(m_addr);
   }
…
…
    // compare peer returns true if lhs is better than rhs
   if (m_peer_selector != false && m_type != 3)
   {
         if (candidate != -1
         && compare_peer_for_peerSelector(*m_peers[candidate], pe)) continue;
    }

   // Below biasing code already exists. Add compare_peer_for_peerSelector before this
   // compare peer returns true if lhs is better than rhs. In this case, it returns true if the
current
    //candidate is better than pe, which is the peer m_round_robin points to. If it is, then just
   // keep looking till you find a better one or the list ends
   else if (candidate != -1
         && compare_peer(*m_peers[candidate], pe, external_ip)) continue;
…
…
   // save prefered list and return connection candidate
   if (m_peer_selector != false)
   {
      fprintf(fp2, "\n\n%s          ", "Prefered List");
      for (policy::iterator i = m_peers.begin();i != m_peers.end(); ++i)
      {
         fprintf(fp2, "%s", (*i)->ip().address().to_string().c_str());
```

```
        }
        policy::iterator i = m_peers.begin() + candidate;
        fprintf(fp2, "\n\n%s %s \n", "Selected Candidate - ", (*i)-
>ip().address().to_string().c_str());
        fprintf(fp2, "%s    \n", "=============================================");
    }
}
```

7. Experimental results

This chapter describes the experiments done to test the functionalities of PeerSelector and to evaluate the impact of the framework when integrated with BitTorrent.

The hardware and platform used for experiments are presented in the first section. Then the environment in which PeerSelector was tested is presented. Then, the various software tools used to carry out the experiments are described. The next section presents scenarios and the setup required to perform experiments. The intention of such scenarios is to assess the correctness of the standalone PeerSelector application but also to test the integration of PeerSelector with an existing P2P system. Then in the last section, some results on BitTorrent downloading performance are provided.

To summarize, there are two main purposes behind the experiments: to test the functionality of the framework, and obtain some performance results in terms of download speed when PeerSelector is integrated with BitTorrent protocol.

7.1 Experimental procedure

The experiment is conducted on an Ubuntu-based machine. The hardware and the operating system used for conducting the experiment are listed in Table 7.1.

Laptop	Memory	Processor	Operating system	Connectivity
Sony Vaio	2 GB	Intel(R) Core(TM) i3 CPU M 350 @ 2.27 GHz	Ubuntu 10.04(lucid)	Eth/Wifi

Table 7.1 Hardware and OS used during experiment

7.1.1 Environment

During the first part of the experiment, the functionalities of PeerSelector are tested. In order to do so, a list of peers which can be supplied either by a tracker, DHT or PEX was needed. For the purpose of the experiment, a tracker was used to get the list of peers in the swarm. Two different torrent files from two different categories were used, to get the diversified set of users. Description of torrent files is shown in Table 7.2.

The same set of torrent files were also used to measure user experience in terms of download speed. As previously mentioned, Bittorrent client named *client_test wa*s modified to bias the default peer selection mechanism, implemented in the libtorrent rasterbar library.

torrent file	BitTorrent Index site	Size of content	Seeds	Leechers	Client
Hindi(Indian) Language(torrent 1)	isoHunt	690.32 MB	21	10	BitTorrent client
English Language(torrent 2)	mininova	161.74 MB	15	5	BitTorrent client

Table 7.2 Description about the torrent files used at the time of experiment

7.1.2 Testing tools

In order to conduct the tests, several tools were used. Browser used for the test is Mozilla Firefox version 3.6.24. Mozilla was used to download torrent files from the BitTorrent search engines: Isohunt and Mininova. PeerSelector uses *libtorrent-rasterbar-0.15.9* library and a sample BitTorrent client named *client_test that* comes with the library. At this point, *client_test* was already modified to integrate PeerSelector functionalities; the modification in the client code is already presented in section 6.4.

7.2 Setup and results

7.2.1 Scenario 1 - Unit testing

Goal

Goal of this scenario is to test proper functioning of the standalone PeerSelector.

Description

PeerSelector should be able to add peers to the database, sort and group the peers based on the preferred interest, and delete peers from the database when required. These three behaviors were tested by executing the Unit Test Application.

Setup

The laptop mentioned in Table 7.1 was used to perform the preliminary tests of implementation, using a unit test file. Test code can be found in Appendix B.6. After running the unit test file, SELECT query was run on the database to verify the operation. See Appendix F.20 for the compilation and linking commands. The unit test file and the MySQL server were run on the same machine.

Results

Each method was tested during Unit testing. Results are shown in Figure 7.1 to Figure 7.4.

A. In Figure 7.1, results of *addpeer* API call are shown. It can be seen that all the IP addresses along with port numbers, info hashes and ASNs information is added to the database. In order to test *addpeers* method, three peers were added in a vector and instead of calling *addpeer* one by one, *addpeers* was called to add all three peers in one run. Refer Figure 7.3 where number of rows are 12 as three peers are added to the table.

Figure 7.1 Scenario 1 - addpeer API call

B: In order to test *getpeers* API's functionalities, unit test application was run. Policy type 0 was passed from command line. Screenshot of the *UnitTest* application's command prompt is shown in Figure 7.2.

In Figure 7.3, score table which stores the information about peer's geo-location is displayed. After comparing the output shown in Figure 7.2 and Figure 7.3, it can be said that the peer clustering based on score type is working correctly.

RTT table and AS Hop table are shown in appendix C.1 and C.2 respectively.

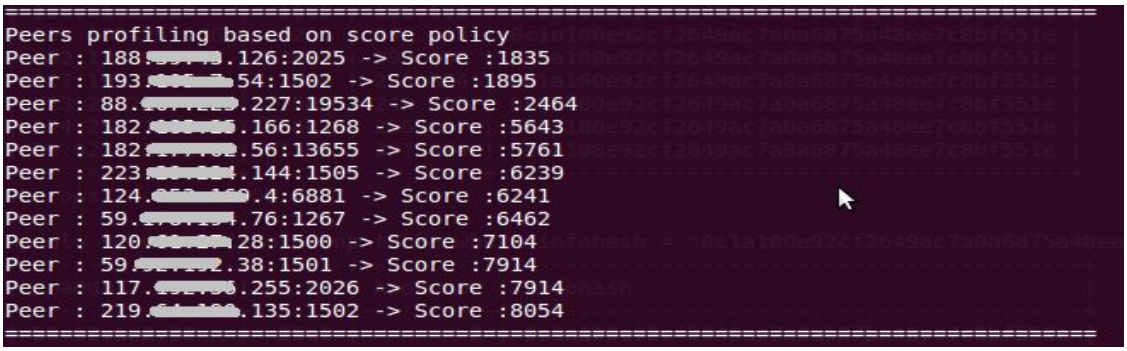

Figure 7.2 Scenario 1 - getpeer API call

```
mysql> select * from score_table where infohash = "0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e" order by score;
+----------------+-------+-------+------------------------------------------+
| ipaddr         | port  | score | infohash                                 |
+----------------+-------+-------+------------------------------------------+
| 188.[redacted].126 |  2025 |  1835 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 193.[redacted].54  |  1502 |  1895 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 88.[redacted].227  | 19534 |  2464 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 182.[redacted].166 |  1268 |  5643 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 182.[redacted].56  | 13655 |  5761 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 223.[redacted].144 |  1505 |  6239 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 124.[redacted].4   |  6881 |  6241 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 59.[redacted].76   |  1267 |  6462 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 120.[redacted].28  |  1500 |  7104 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 117.[redacted].255 |  2026 |  7914 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 59.[redacted].38   |  1501 |  7914 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 219.[redacted].135 |  1502 |  8054 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
+----------------+-------+-------+------------------------------------------+
12 rows in set (0.01 sec)
```

Figure 7.3 Scenario 1 - Table that stores score value

C: This Unit test was conducted to test the functionalities of the delete peer method. A peer from score table, having IP address "182.177.62.56", port number 13655, infohash "0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e" was passed as an argument to *deletepeer* API. It can be clearly seen in the Figure 7.4 that the specified peer is deleted from the table and the total number of rows in table is now 11 instead of 12. The *deletepeer* API was tested for all the policies and it was successfully able to delete the peer from the database however, to avoid the repetition it is not shown in this master thesis report.

```
mysql> select * from score_table where infohash = "0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e" order by score;
+----------------+-------+-------+------------------------------------------+
| ipaddr         | port  | score | infohash                                 |
+----------------+-------+-------+------------------------------------------+
| 188.[redacted].126 |  2025 |  1835 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 193.[redacted].54  |  1502 |  1895 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 88.[redacted].227  | 19534 |  2464 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 182.[redacted].166 |  1268 |  5643 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 223.[redacted].144 |  1505 |  6239 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 124.[redacted].4   |  6881 |  6241 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 59.[redacted].76   |  1267 |  6462 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 120.[redacted].28  |  1500 |  7104 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 117.[redacted].255 |  2026 |  7914 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 59.[redacted].38   |  1501 |  7914 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 219.[redacted].135 |  1502 |  8054 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
+----------------+-------+-------+------------------------------------------+
11 rows in set (0.00 sec)
```

Figure 7.4 Scenario 1 - deletepeer API call

7.2.2 Scenario 2 - Integration testing

Goal

The objective of this test case is to verify the integration of libtorrent with the framework.

Description

The objective of this scenario is to evaluate if PeerSelector was integrated properly with libtorrent client or not. After the integration, the libtorrent-based client *client_test* should be able to call public APIs of PeerSelector.

Setup

In order to test this scenario, *BitSwiftSelector.cpp* was copied inside *~/libtorrent-rasterbar-0.15.9/src* folder. Then *BitSwiftSelectot.hpp* file was copied to *~/libtorrent-rasterbar-0.15.9/libtorrent/include* folder. The Makefile was changed to compile *BitSwiftSelector.cpp* and to link it with libtorrent's library. The commands for compilations and linking can be found in Appendix F.20.

Results

A: One of the objectives of the project was to integrate PeerSelector framework with BitTorrent P2P system. In order to test if the framework is integrated properly with BitTorrent, BitTorrent client application was run with *PeerSelector* functionalities. The *client_test* application was successfully able to call *PeerSelector* APIs and the results are displayed on the client terminal, matching with the value stored in the database. List of peers returned by the tracker is shown in Figure 7.5. These lists of peers are supplied to *PeerSelector* framework for profiling. In Figure 7.6, a sorted list of peers according to geo-location is shown.

Figure 7.5 Scenario 2 - List of peers returned by tracker

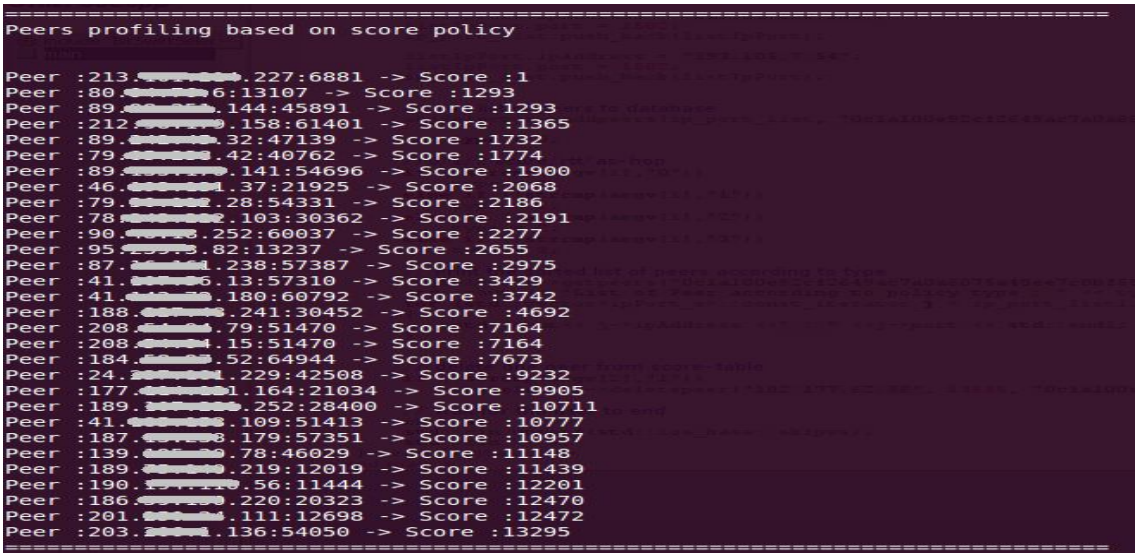

Figure 7.6 Scenario 2 - Sorted list of peers returned by PeerSelector

It is clear from the experiment that BitTorrent client is integrated properly with framework and is able to add peers to PeerSelector's database using *addpeers* API. Similarly, client is also able to call *getpeers* API to get the preferred list of peers returned by PeerSelector.

7.2.3 Scenario 3 - System testing

7.2.3.1 BitTorrent performance measurements

Goal

The goal of this test is to obtain BitTorrent performance results in terms of download speed and compare the user experience when BitTorrent is integrated with PeerSelector with the experience when the application is not integrated.

Description

After making sure the framework was integrated properly and working fine, performance of BitTorrent in terms of download speed were to be assessed next. Objective of this section is to test the system as a whole and obtain the results in terms of download speed. It is also worth mentioning that the laptop involved in this test was not performing any other CPU or network related tasks, which could influence the results.

In this experiment, the behavior of libtorrent, when integrated with PeerSelector, under four different policies was tested. Tests were conducted in three different environments.

In the first environment, BitTorrent client run with four different policies, each for 3 minutes. Also, approximately 1 minute of switching time between each policy was considered. So, in total, a 4 minute period was assigned to each policy.

Since BitTorrent application runs on open Internet, the environment is not stable. Peers might join or leave the swarm anytime. To minimize the impact of faster peers leaving or joining the swarm, all four policies were run within a time span of 16 minutes, assuming that peers in the swarm would not vary drastically within 16 minutes. It was an attempt to execute all 4 policies with as much of the same set of peers as possible.

Furthermore, in the second experiment run, PeerSelector was tried with the BitTorrent client using another torrent file. This torrent file was more global in nature and its peers were across the globe.

The third experiment was performed in a similar way however, in this case, each policy was tested against the random policy. The idea behind setting up this environment is to nullify the impact of faster peers joining the swarm in a particular policy and possibly leaving the swarm in the other policy. In other words, in this environment the behavior of the integrated system, without considering the external factors, was tested. External factors could be different peer sets provided by the tracker at two different timestamps, or dissimilar set of peers were available for connection when seeding with different policies.

Setup

To setup this scenario, first it was required to compile the *client_test* client using the command shown in Appendix F.21.

The following steps describe the taken procedure:

- A torrent file was downloaded from IsoHunt or Mininova. Name of the file is not mentioned for copyright reasons.
- The following command was executed on the terminal.

```
./client_test noName.torrent policyType urgentFlag
```

  In the command, policy type could be 0, 1, 2, or 3, depending on the chosen policy. urgentFlag can be 0 or 1: 0 stands for calculating the preference value at run time and returning the new set of peers; 1 stands for using the peers grouped on previously calculated value.
  Firstly, the performance of libtorrent was tested by enabling score policy. In the subsequent run, policyType flag was changed to 1, 2, and 3 for the other three policies.
- To display information on the terminal one can use “i” option on the terminal. After 3 minutes of seeding q was pressed to pause the download.
- Log files were stored by name peers_log_0_run1 and PreferredList_0_run1. Here, 0 stands for score policy.
- The command mentioned in step two was run for three times in a row with policyFlag 1, 2, and 3, with an interval of 3 minutes in each case.
- For each policy, the average download rate was calculated by running a simple shell script on the peers_log_x_run1 file where x = 0,1,2,3. Script displays the average download rate on the terminal.

- Later, in the second run, urgentFlag was changed to 1 to save calculation time. In this case, historical values were used for peer clustering.

In the second environment, the following steps were followed.

- In this environment, the same steps as in the previous environment were followed, however, the *client_test* application was run with a more global torrent.

```
./client_test noName2.torrent policyType urgentFlag
```

- Logs were collected and parsed to get the average download speed in each case.

Setup for the third environment is explained below.

- In the third test, 2 instances of the same application were run at a time. On one hand, application was run with less distance, low latency or less AS hop policy. On the other hand, same application was run with random policy.
- Logs were collected and parsed to get the average download speed in each case.

Download speeds obtained in each policy for each environment were compared with the random download speed. PreferredList_0_run1 was parsed to check if PeerSelector is returning the correct candidates for establishing peer-to-peer connections.

Results

A. Download performance of torrent 1 is shown in Figure 7.7. Download time in seconds is shown on x-axis and the download rates in KB/s are shown on y-axis. After collecting 150 samples, they were averaged to calculate the average download speed for all four policies. It can be clearly seen that average download speed when random policy is enabled is higher than all other policies. This is because of the fact that two faster peers, one from India and another one from United Arab Emirates (AE), had joined the swarm after around 1 min and 30 sec of seeding when random policy was enabled.

When AS-Hop policy was enabled, one of these two peers – with 2 AS-hops away from our peer --- was selected and thus a higher download rate was seen towards the end of seeding. The peers selected in other two policies --- score and RTT -- exhibit a lower download rate. This may come from the fact that peers selected in score policy and RTT policy cases might have limited network bandwidth. It has also been observed that few faster peers --- one from India, one from AE and another from Holland, which were not present at the time when download was happening according to score and RTT policies, have joined the swarm when random policy was enabled.

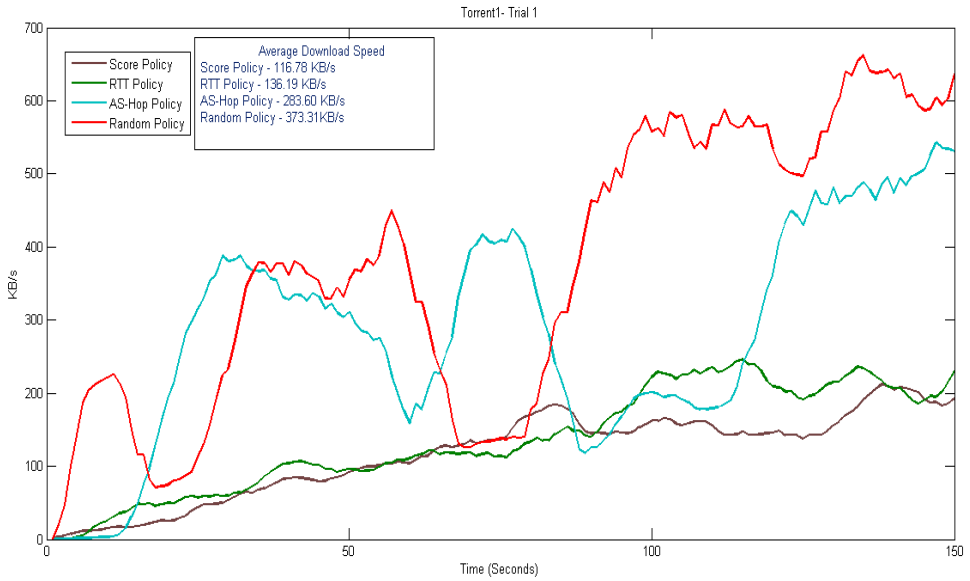

Figure 7.7 Download performance for all four policies

B: In order to come to some conclusion, the same setup was re-run, as shown in Figure 7.8. This time, a mixed graph was revealed. The average download time when RTT policy was enabled is higher than all three policies. Score policy also downloads faster than Random policy in this case.

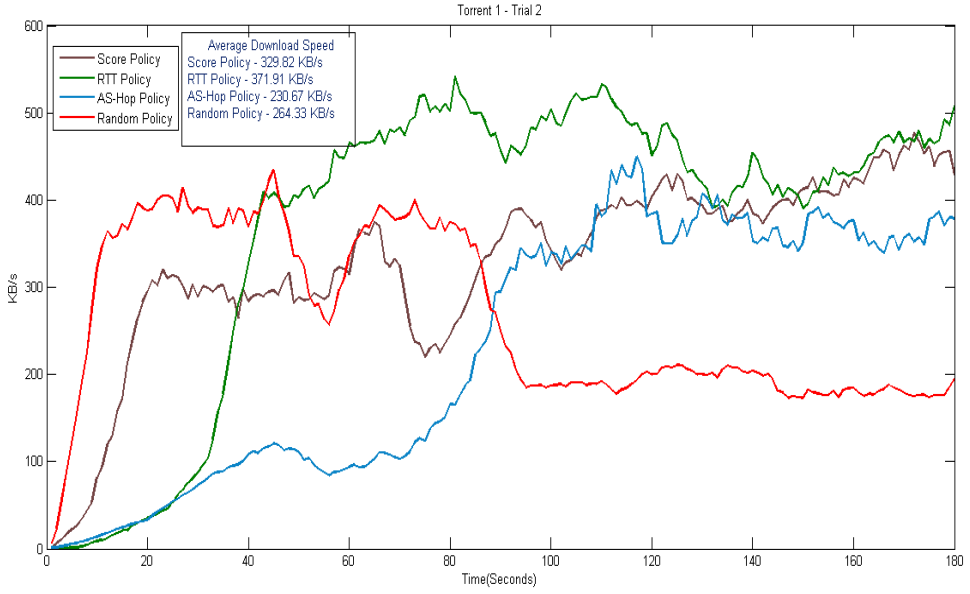

Figure 7.8 Download performance for all four policies in trial 2

The downloading speed was very uneven with random selection policy. After analyzing the log we have found that the reason for uneven graph was: a peer from Romania was uploading pieces at very uneven rate from around 20 KB/s to 300 KB/s. Even after continuing seeding for more than 150 seconds, downloading speed for random policy did not improve.

C: In Figure 7.9, the download rates with all four policies, by keeping the same setup but changing the torrent file are shown. This time a more global content was picked for download, whose peers were across the globe.

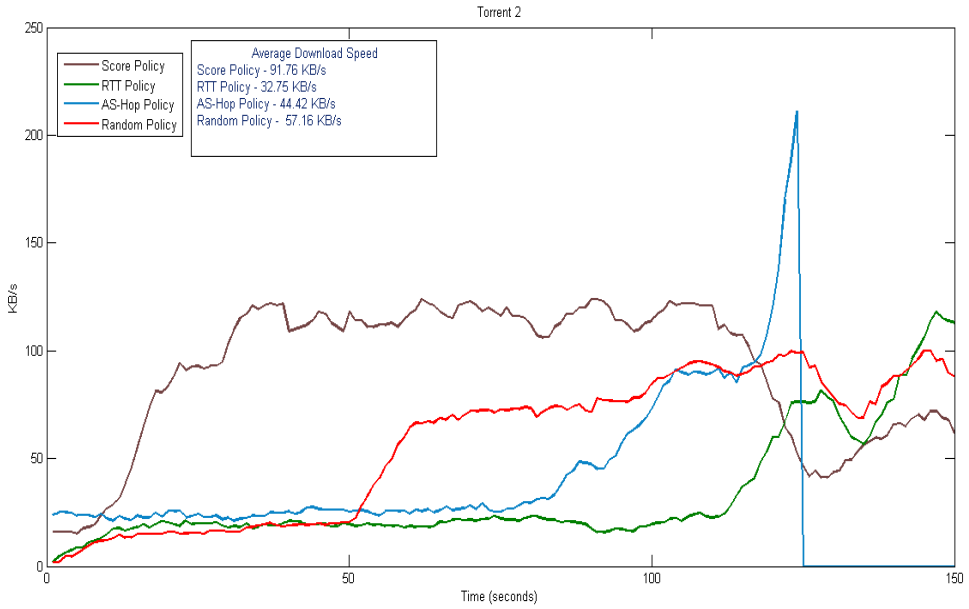

Figure 7.9 Download performance for all four policies with second torrent file

In this case, peers selected using score policy downloaded faster than other three policies. This was because score policy selected nearby peers based on geographic distance and it proved to be faster peers. A sudden increase of the download rate was observed with the AS-Hop policy, after 2 minutes of downloading.

D: Even after choosing shorter download time, it was not clear whether the environment will be same in all four policies, i.e few peers might download in one policy and might stop downloading by leaving the swarm in another policy.

In order to keep the environment similar, instead of a sequential download (one policy at a time), simultaneous download was started by running the two instances of *client_test* application from two different directories. Each policy was tried against random policy. To start with, score policy was tried against random policy. Figure 7.10 shows that score policy does not produce a better result when compared with random policy. RTT-based policy download is shown in Figure 7.11 and it also displays a similar behavior --- the graph is very uneven. A surprisingly low download speed of random policy was observed when it was tried with AS-Hop policy as illustrated in Figure 7.12. We found that, since two instances of BitTorrent client were running locally --- one for score/RTT/AS-Hop policy and another one for random policy --- they were becoming peers of each other and were affecting the download radically.

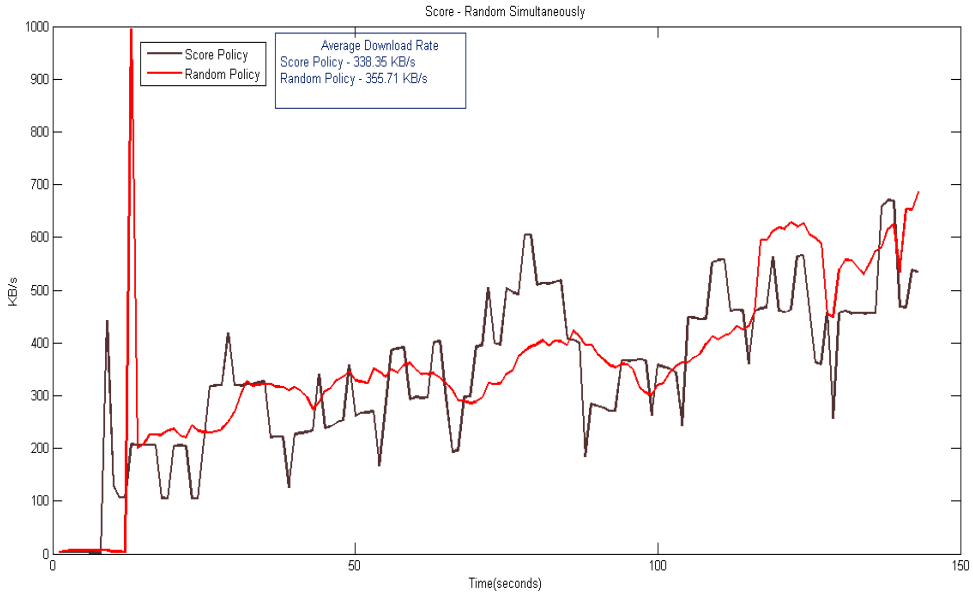

Figure 7.10 Score Policy based peers and random policy based peers

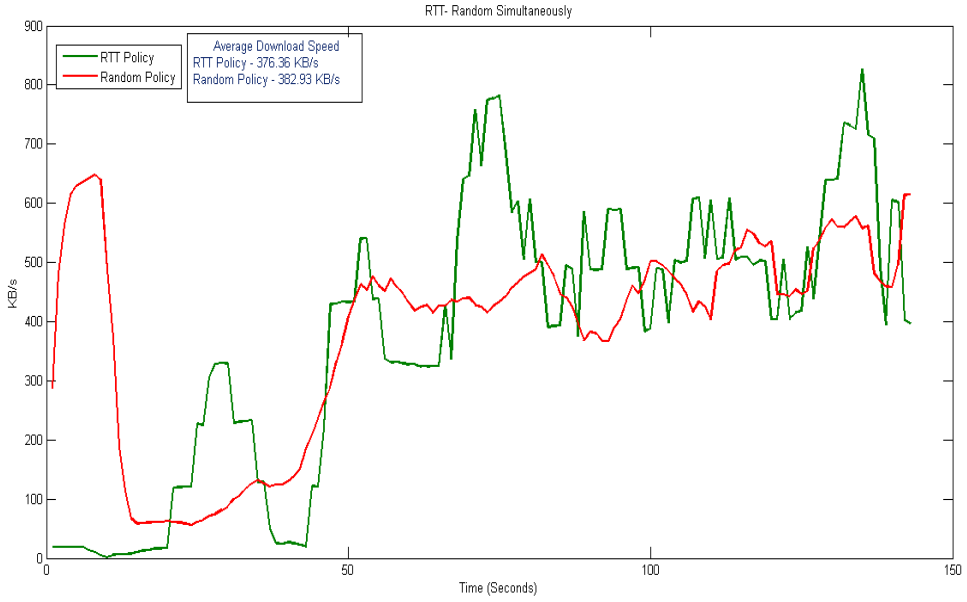

Figure 7.11 RTT policy based peers and random policy based peers

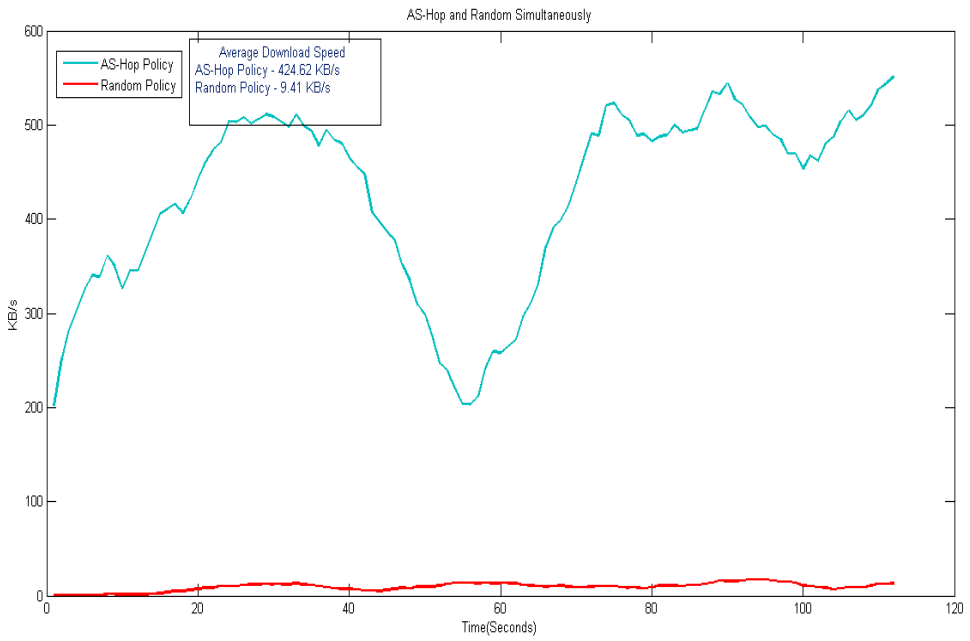

Figure 7.12 AS-Hop policy based peers and random policy based peers

7.2.3.2 Swift performance measurements

Goal

The objective of this experiment is to test the impact of PeerSelector when integrated with Swift protocol. Another objective of this test is to make sure that PeerSelector can be plugged easily with different distributed protocols.

Description

To understand the impact of PeerSelector, we integrated PeerSelector with another P2P distributed protocol --- Swift. The test was conducted in a controlled environment because Swift is not yet deployed in the Internet. In this scenario, more accurate logs were obtained since the setup was controlled and thus peers in the swarm were fixed.

In another ongoing project, PeerSelector was already integrated with Swift using PlanetLab [54] nodes. One of the objectives of the project was to change the default behavior of Swift, where Swift was establishing new connections with each newly discovered peer. In this project, Swift was modified to establish new connections only with preferred peers returned by PeerSelector framework.

Setup

A small size swarm with 12 instrumented nodes running Swift was created. Out of 12 nodes, one node was set up as initial seeder and another node was set up as a peer --- we can call it as requester, who was interested in the content. Initial seeder node was responsible for bootstrapping the requester, who was interested in content, into the swarm. It was running

PEX gossiping algorithm to know about the peers in the swarm. Initial seeder was also responsible for providing all discovered peers to the requester through PEX gossiping. After finding its neighbor, Swift protocol running on the requester informs PeerSelector framework about the newly discovered peers. In the background, framework profiles those peers. The requester then asks the framework for peers based on preferred interest, or requests to delete unwanted peers.

Additionally, for this test, the number of connections that swift establishes with other PlanetLab node was fixed to five.

Moreover, in Swift, content is identified using a root-hash which is different from the info-hash available at torrent indexing sites. Therefore, any publicly available torrent file could not be used for the download. For the experimental purpose, a video file was selected and a small swarm with 12 nodes, as described earlier, was created.

Results

There seems to be some improvement in terms of download speed when the framework is tested with Swift in a manually configured setup in PlanetLab. In two trials, output of another master thesis project, shows that download speed is better when score and RTT-based latency metrics are used compared to other metrics. In case of the score metric, download speed has increased 50% in both trials.

In case of the RTT-based latency metric, download speed has increased around 10% in both trials. In addition, when RTT-based latency metric was used, our peer experienced a 25-30 seconds startup delay because PeerSelector was actively probing each node in the swarm three times and was calculating the mean of those three values.

Furthermore, ASHop-count policy performed better in terms of download speed when compared with random policy in one trial and worse in another trial. With the AS-Hop metric, in one trial, a 35% faster download was experienced and in another trial, a 13% slower download was obtained with respect to random policy.

8. Discussion

The experiments conducted in the previous section served as a tool to evaluate the functionalities of PeerSelector and its impact on download performance for distributed P2P systems. Although we did not present any new metric for peer clustering, experimental results highlighted the potential benefits of using PeerSelector framework.

8.1 Framework functionalities

In this thesis, we proposed a framework which is used for interest based peer clustering. With our initial set of requirements, we built a design which was implemented and tested.

The results from scenario 1 demonstrated that the framework offers a composition of clustering mechanisms. In contrast to the previous proposals – based on individual clustering technique, this thesis is able to integrate different peer clustering techniques into one framework. The results shown in Figure 7.2, appendix C.1, and C.2 demonstrated the functionality of the framework in terms of grouping peers according to geo-location, latency and AS-Hops. Furthermore, the experimental results from scenario 1 also confirmed that the framework met the goal, i.e, PeerSelector is able to add the P2P distributed protocols' notified peers into the database as shown in Figure 7.1, group those peers based on preference interest (refer to Figure 7.2 and Figure 7.3), and delete unwanted peers from the database (refer to Figure 7.4).

In addition to these, during the implementation phase we added an API named *compareBasedOnPS*, which is able to compare two peers based on user's indicated interest and returns the better peer to the user. It was required at the time of integrating our framework with BitTorrent protocol, where the peer (requester) wanted to find a peer with whom it could establish a TCP connection. One more API--- *getPeersImmediately* --- was added to save the startup time without spending time in calculating peer's information.

Having experimented and proved that the framework worked in the way it was designed to work, it does not mean that it cannot still be optimized in the future. The performance of the RTT-based latency computation raised some concern as the requesting peer was experiencing a longer startup time (known as the learning period), in turn, increasing total download time. This behavior was observed at the time of testing the framework with BitTorrent. The unreachable nodes were significantly increasing the learning period as the requesting peer was waiting for timeout to figure out the reachability of the node. There are several reasons why nodes cannot be reached --- 1) they might be behind NATs or firewalls, 2) they might not be alive 3) they might have left the swarm. In future, instead of actively probing the nodes, some latency prediction system should be evaluated.

Furthermore, we evaluated that the framework proposed and implemented in this thesis is portable across different distributed P2P system with minor code modifications. For example, Figure 7.5 and Figure 7.6 show that the framework was integrated successfully with a deployed BitTorrent P2P system. In addition, framework was also tested with Swift application. However, the framework opens much scope for testing. In future, other distributed P2P systems and many latency sensitive applications should be integrated and tested with the framework.

Additionally, when compared with previously proposed architectures like P4P architecture [19] and Oracle based architecture [14], our framework does not require any additional infrastructure support. As discussed earlier, P4P architecture requires changes at the application side and also at the ISP side. Similarly, techniques such as BNS [18] also require changes either at the tracker or at the ISPs. However, this thesis introduced the concept of moving peer selection functionalities to the client side and providing flexibility to the client to group peers without modifying external entities like trackers or ISPs.

8.2 Framework performance impact

Another important consideration was to analyze the impact of framework on the download performance of the distributed P2P systems. When integrated with a deployed BitTorrent system, a mixed download speed was observed. In one trial, as shown in Figure 7.7, peers selected using score and RTT policies were not performing well compared to random and AS-Hop policies. This was from the fact that peers responsible for increasing the download rate in case of random and AS-Hop policy were not present at the time when score and RTT policies were enabled. In another trial, it was observed (Figure 7.8) that RTT policy was performing better, in terms of download speed, than other three policies. It was also observed that the download speed was very uneven when peers were selected using random policy. This was from the fact that a peer from Romania was uploading pieces at very uneven rates. Furthermore, it was observed that the peers selected using score policy performed well and provided an enhanced download speed in 75% of the trials.

Further, when downloading from a torrent with global content, whose peers were across the globe (Figure 7.9), it was observed that the peers grouped using score policy performed better than other three policies, more importantly better than random policy. In this case, a geographically closer peer recommended by PeerSelector was a faster peer and was not chosen when random policy was enabled. As shown in Figure 7.9, a sudden increase of the download rate was observed with the AS-Hop policy, after 2 minutes of downloading. This was from the fact that the peer responsible for increasing the download rate was from India and was not suggested by PeerSelector framework during the start of seeding as it was far from our peer in terms of AS-Hops.

As discussed earlier, there are many reasons for not overly promising results. To summarize, for example in trail 1, random peers experienced a better download speed because two peers

were randomly picked and they happened to be faster peers---one from India and another one from AE. Another reason for slow download rate in case of AS-Hop and RTT policies could be because peers selected using AS-Hop policy and RTT policy might have limited network bandwidth. With this finding we believe the framework presented here should, in the future, be enhanced to add network bandwidth as one of the metrics, which can be used independently or can be used in combination with other metrics to generate complex queues ---- queues which can contain peers from all other queues. It is also important to mention that in some cases a particular peer uploaded content at very uneven rates and this lead to an uneven graph. In order to minimize the implications of such peers some work should be done in this area. For example, actual download rates along with standard deviation in download rates can be used as an evaluation metric.

Additionally, there is substantial room for improvement on the experiments side. During the testing with BitTorrent protocol (Figure 7.7 to Figure 7.12), it was discovered that the peers recommended by PeerSelector are only a fraction of the entire set involved in the download. It would be interesting to investigate the download performance of BitTorrent when all the peers involved in the download are suggested by PeerSelector. Additionally, it would be interesting to perform an experiment with limited number of peers recommended by PeerSelector, i.e, by limiting the number of download slots because some of the peers recommended by the BitTorrent are either very slow or non-responsive.

On the other hand, a good improvement was seen when framework was integrated with Swift. The peers selected using the score policy have shown a two fold increase and the peers selected using the RTT-based latency metric have shown a 10% increase in download speed, compared to random policy. The peers grouped according to the AS-Hop count metric also resulted with a 35% performance increase in one experimental trial however, in another trial, a 10% decrease was seen.

It was observed that because of the limited number of slots available for connections, Swift established all connections with the peers recommended by PeerSelector. We believe that this is one of the reasons why Swift performed better than BitTorrent when integrated with PeerSelector, in terms of download speed.

9. Conclusion

In this master thesis we presented the design, implementation and evaluation of a framework called PeerSelector, used for peer clustering according to predefined criteria. In particular, the framework supports clustering of peers according to geographic closeness, low latency and nearby ASes.

The modular and generic API of the framework makes PeerSelector portable and easily deployable with any distributed P2P system. Unlike previous approaches, our framework does not depend on any external entity or additional Internet infrastructure to function correctly. More specifically, PeerSelector runs at the end-user side and is used to group peers according to the user's indicated interest or preference. The current implementation of our framework supports three predefined options --- user's interest or preference --- that a user can select to group peers.

For developers, PeerSelector is a framework that can be easily extended, customized, and integrated with various applications. New policies for peer clustering can be added, without changing API interfaces. In addition, the framework is free and the source code can be found at the GitHub repository [36]; it can be used or modified for any purpose.

To illustrate the flexibility of the framework, PeerSelector was integrated and tested with a DHT implementation, as well as, BitTorrent and Swift P2P protocols using generic interface handlers. However, new interface handlers may be added with minor adjustments to the framework API. The obtained results from the integration confirmed that PeerSelector groups peers according to the user specified policies and communicates with the integrated protocols as anticipated.

Preliminary results on download performance when PeerSelector was integrated with the BitTorrent system are not conclusive. In experiment trials with policies that group peers according to low latency and AS-hop counts, a reduced download speed has been observed, compared to the scenario where peers are randomly selected. On the other hand, grouping peers according to geographic closeness more consistently shows enhanced download speed for all scenarios.

Preliminary results on download speed from the integration of PeerSelector with Swift are more consistent. In particular, when peers are grouped by geographic closeness, download speed observed is twofold better than when peers are randomly selected. When peers are grouped according to low latency, approximately a 10% increase in download performance has been observed. Finally, grouping peers according to nearby ASes resulted with a 35% performance increase in one experimental trial and a 10% decrease in another trial.

Though the preliminary results on download performance are not as promising, this master thesis project aimed to be a first attempt at integrating previous peer clustering mechanisms into one flexible framework that can be deployed with different P2P systems and where users can specify their preferred interest on peer clustering.

10. Future work

There are many open possibilities for future work. In this section, open issues and areas of improvement are suggested.

10.1 Support for streaming applications

As future work, we propose to enhance the framework to accommodate video-on-demand VoD streaming applications. The locality-biasing becomes important for minimizing the delay and for providing quality video. In VoD service, creating group and selecting servers in a fashion that considers locality information, latency and AS-Hops can improve performance in term of user experience, such as startup delay and piece picking continuity.

10.2 Online multiplayer gaming and other latency sensitive applications

As explained, this framework was tested with BitTorrent and Swift applications. However, it is not tested with latency sensitive applications. For online gaming, high lag between peers is not desirable [4]. These latency sensitive applications can take advantage of the framework to find low latency peers.

10.3 Support for more policies

This project was an attempt to integrate different locality biasing approaches in one framework. Currently, four policies are supported, however, the framework can support more policies. For example, a peer can select its neighbors according to the bandwidth between the peer and its neighbors. So, in the future, more policies can be added.

10.4 Support for creating complex queues

Current implementation supports four queues. However, new queues can be derived from these queues. Framework can support complex queues. As an example, a user can specify number of peers from each queue and PeerSelector should be able to add those peers in a "derived queue". Peers from all four queues can be added in this queue.

10.5 Performance improvement of the framework

Current implementation takes longer startup times when the RTT-based latency metric is used (known as the learning period), in turn, increasing total download time. Some work can be done in this area to reduce the learning period. For example, instead of actively probing the peers some latency prediction system should be evaluated. An alternative latency prediction system called *Htrae*[4] has been proposed for game matchmaking.

10.6 User experience when experimented with BitTorrent

When the framework was integrated with BitTorrent system, user's experiences in terms of download performance were not so promising. The important topic that would be interesting to investigate is the download performance of BitTorrent when all the peers involved in the download are PeerSelector's suggested peers. It would also be interesting to study the behavior of BitTorrent by limiting the number of download slots because some of the BitTorrent discovered peers are very slow or sometimes non-responsive. So, an experiment with a scenario where only a limited number of peers preferred by PeerSelector is used should be performed.

10.7 PeerSelector GUI

To make the framework user-friendly, a GUI can be developed. Users can specify their choice through the graphical interface. For example, their preference interest, number of peers, history-prioritization and so forth.

Appendix A

Design diagrams

Appendix A includes the diagram used during design phase. It include class diagram and sequence diagram. Figure A.1 is the PeerSelector class diagram. Figure A.2 to A.5 are sequence diagrams. These diagrams are generated using Microsoft visio 2010[35] and dia [29].

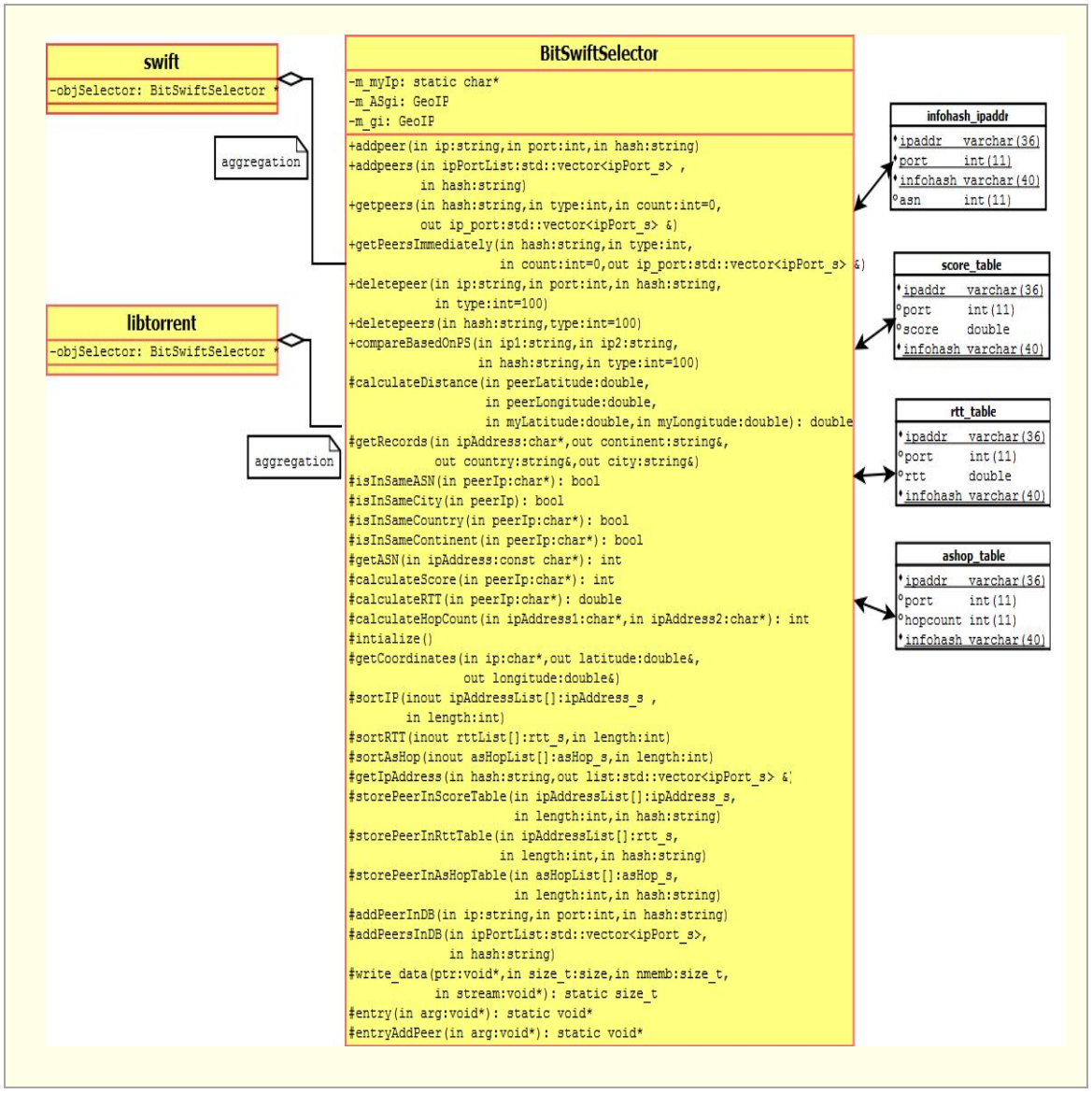

Figure A.1 PeerSelector - Class Diagram

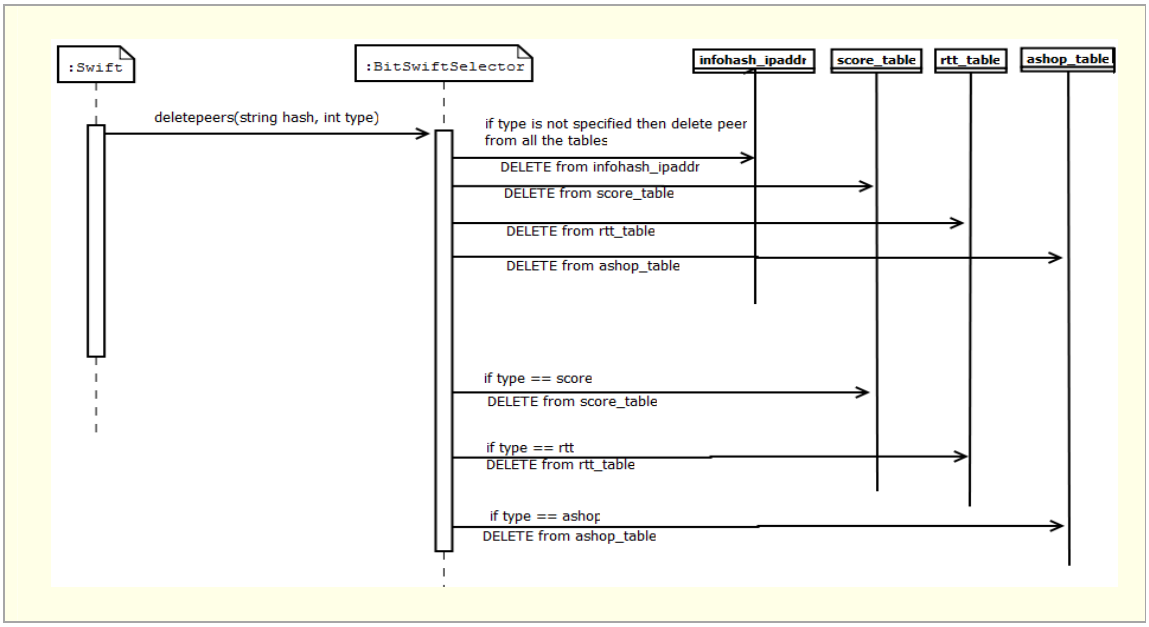

Figure A.2 Sequence Diagram - deletepeers

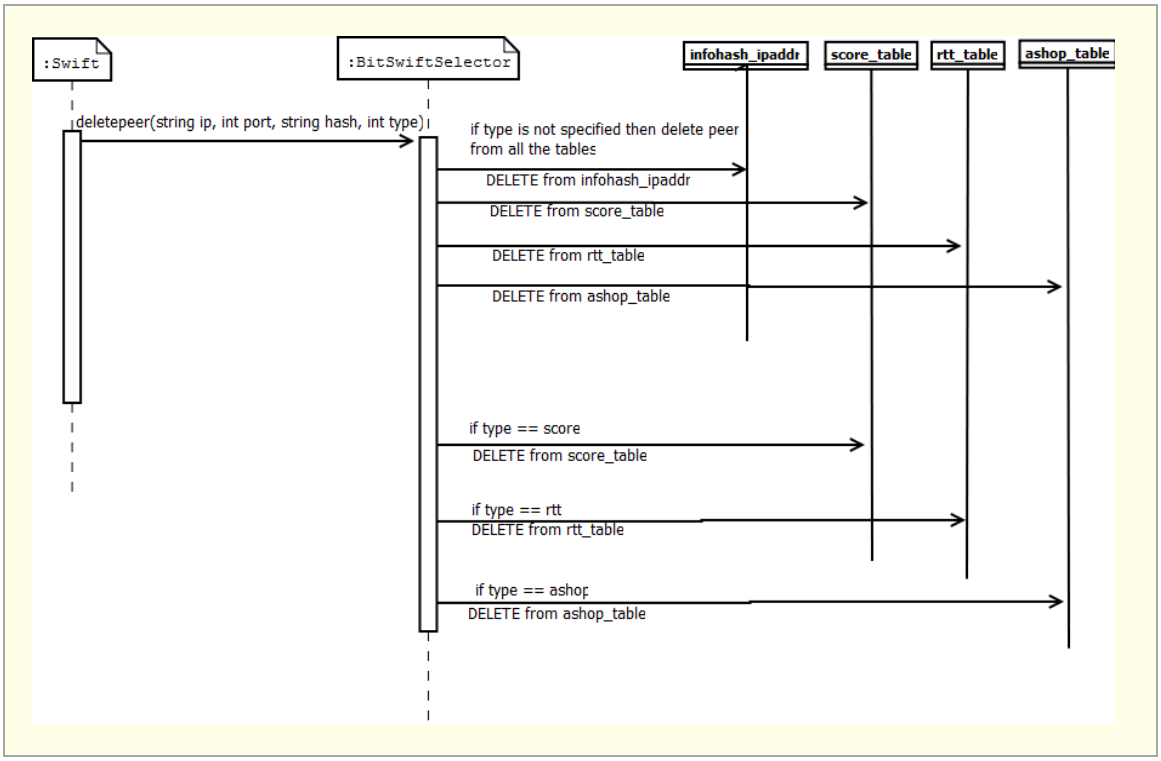

Figure A.3 Sequence Diagram - deletepeer

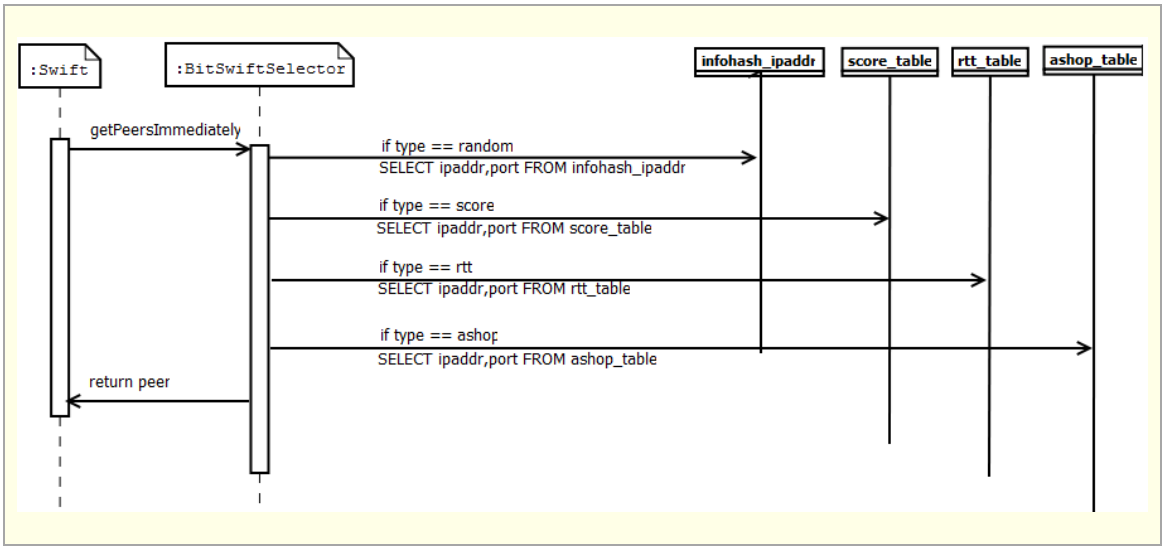

Figure A.4 Sequence Diagram - getPeersImmediately

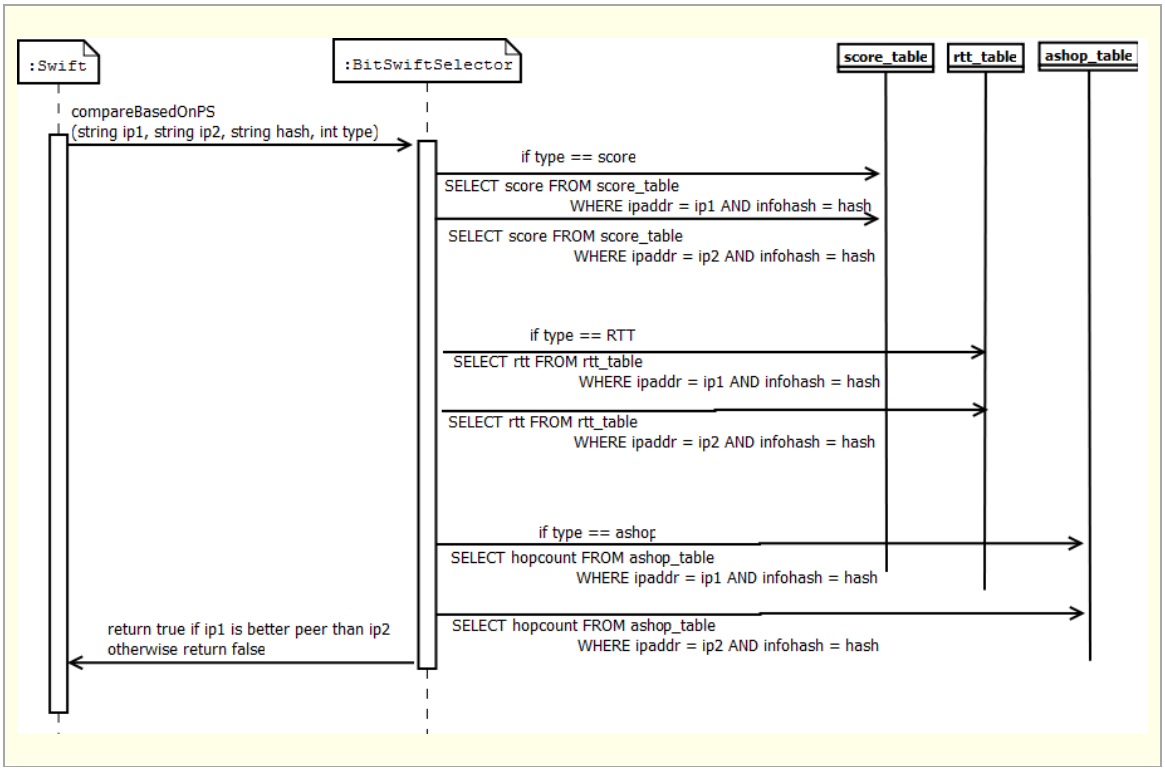

Figure A.5 Sequence Diagram - compareBasedOnPS

Appendix B

Source code

Listing B.1 Working with pymdht - Installation

pymdht[11] is the flexible implementation of main line DHT. It is developed by KTH PHD students Raul Jimenez et al. The code base available at GitHub [11], is used for developing the PeerSelector prototype. Steps followed for developing the prototype are as follows.

- pymdht package available at GitHub[11] was installed
- Then below command was run on the terminal

  $ *./run_pymdht_node.py*
  Using the following plug-ins:
  ** plugins/routing_nice_rtt.py*
  ** plugins/lookup_a4.py*
  ** core/exp_plugin_template.py*
  Path: /home/raul/.pymdht
  Private DHT name: None
  debug mode: False
  bootstrap mode: False

  Type "exit" to stop the DHT and exit
  Type "help" if you need
  help

  Available commands are:
  - help
  - fast info_hash bt_port
  - exit
  - m Memory information

- *fast info_hash bt_port* was type on the terminal
- The list of IP addresses and port numbers were passed to the Test Application in Listing B.4
- Test application was successfully able to sort those peers according to Geographic distance using *calculateScore* method.

Listing B.2 Working with pymdht - parse.sh Parser for parsing pymdht output

```
#!/bin/sh
egrep
'[[:digit:]]{1,3}\.[[:digit:]]{1,3}\.[[:digit:]]{1,3}\.[[:digit:]]{1
,3}' < input.txt | tr -d "'[]" | sed -e 's/), /)\n/g' | tr -d "()" |
tr -d " " > output.txt
```

Listing B.3 Working with pymdht - dht.sh - run pymdht

```
#!/bin/sh
cd ~/Desktop/arno/pymdht/
./run_pymdht_node.py<<INPUT
fast $1 0
INPUT
```

Listing B.4 Working with pymdht - Test application

```
int main(int argc, char *argv[])
{
     // run dht.sh which inturn runs pymdht
    char cmd[50] = "/bin/sh ./dht.sh ";
    strcat(cmd, argv[1]);
    strcat(cmd, " > input.txt");
    system(cmd);

    // parse the output from the dht and retrieve peers IP address
and port number
    system("/bin/sh parser.sh");
    parser();
  //   store   peers   ipaddr,   port   number   and   infohash   into
infohash_ipaddr table as described in Listing 6.7
…
…
// calculateScore method is used to calculate the score of each peer
from  our  peer  and  then  sorted  list  of  peers  were  stored  in
score_table
}

//  this  method  is  used  to  read  output.txt  file  generated  after
parsing
void parser()
{
     ipPortList =  new ipPort_s[100];
```

```
      std::string line;
      std::ifstream file("output.txt");
      if (!file)
      {
      cout << "Error: Cannot open file" << endl;
      }
     // read file line by line
      while(std::getline(file, line))
      {
      std::string token;
      std::istringstream tokens(line);
      int count = 1;
     // get IP address and port number separated by ,
     // store ip, port in ipPort data structure
      while(std::getline(tokens, token, ','))
      {
            if (count == 1)
            {
                  ipPort.ipAddress = token;
                  count++;
            }
            else if (count == 2)
            {
                  ipPort.port = atoi(token.c_str());
            }
      }
      ipPortList[length] = ipPort;
      length++;
      }
}
```

Listing B.5 Implementation initialize.sh

```
#!/bin/sh
PACKAGE_PATH=`pwd 2>&1`
export CLASSPATH=$PACKAGE_PATH/as-distances-
1.0/se/sics/asdistances:.
cd as-distances-1.0
javac -d . ASHop.java
```

Listing B.6 Testing UnitTest.cpp

```
// Unit test file to test add, get and delete functionalies
#include "BitSwiftSelector.hpp"
```

```
int main(int argc, char* argv[])
{
      // Stores IP address and port number of preferred peers
      std::vector<ipPort_s> ip_port_list, ip_port_list1;
      ipPort_s listIpPort;

      // Object of BitSwiftSelector class to access its interface
      BitSwiftSelector *objSelector = new BitSwiftSelector();

      if (objSelector)
      {

      // Test Add Peer method. one by one, pass IP address, port
number and hash
      objSelector->addpeer("188.39.43.126",                    2025,
                "0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e");
      objSelector->addpeer("88.167.225.227",                  19534,
                "0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e");
      objSelector->addpeer("182.177.62.56",                   13655,
                "0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e");
      objSelector->addpeer("182.185.25.166",                   1268,
                "0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e");
      objSelector->addpeer("117.192.36.255",                   2026,
                "0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e");
      objSelector->addpeer("120.61.27.28",                     1500,
                "0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e");
      objSelector->addpeer("124.253.169.4",                    6881,
                "0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e");
      objSelector->addpeer("59.178.194.76",                    1267,
                "0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e");
      objSelector->addpeer("223.29.224.144",                   1505,
                "0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e");

      // Test addpeers method. Store all the IP addresses in a vecor
and pass entire vector
      listIpPort.ipAddress = "59.92.192.38";
      listIpPort.port = 1501;
      ip_port_list.push_back(listIpPort);

      listIpPort.ipAddress = "219.64.190.135";
      listIpPort.port = 1502;
      ip_port_list.push_back(listIpPort);

      listIpPort.ipAddress = "193.105.7.54";
      listIpPort.port = 1502;
      ip_port_list.push_back(listIpPort);
```

```
     // add list of peers to database
     objSelector->addpeers(ip_port_list,
                     "0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e");

     int type = 0;

     //0/1/2 score/rtt/as-hop
     if (!strcmp(argv[1],"0"))
          type = 0;
     else if (!strcmp(argv[1],"1"))
          type = 1;
     else if (!strcmp(argv[1],"2"))
          type = 2;
     else if (!strcmp(argv[1],"3"))
          type = 3;

     // Print the sorted list of peers according to type
     objSelector->getpeers(
     "0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e", type,
     ip_port_list1);
     std::cout << "List of Peer according to policy type : " <<
     type << std::endl;
     for(std::vector<ipPort_s>::const_iterator j =
            ip_port_list1.begin(); j != ip_port_list1.end(); ++j)
     {
         std::cout<< j->ipAddress <<" : " <<j->port << std::endl;
     }

     // get peers based on history
     objSelector->getPeersImmediately(
     "0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e", type,
     ip_port_list1);
     std::cout << "List of Peer according to policy type : " <<
     type << std::endl;
     for(std::vector<ipPort_s>::const_iterator j =
            ip_port_list1.begin(); j != ip_port_list1.end(); ++j)
     {
         std::cout<< j->ipAddress <<" : " <<j->port << std::endl;
     }

     // delete one peer from score-table
     if (!strcmp(argv[2],"1"))
          objSelector->deletepeer("182.177.62.56",           13655,
"0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e", 0);

     // wait for the user to end
```

```
      char a;
      std::cin.unsetf(std::ios_base::skipws);
      std::cin >> a;
      }
}
```

Listing B.7 Calculate geographic distance in kilometers

```
// API for calculating distance
double BitSwiftSelector::calculateDistance(double peerLatitude,
double peerLongitude, double myLatitude, double myLongitude)
{
    // get distance on Y axis
    double yDistance = abs(myLatitude - peerLatitude) *
NAUTICALMILEPERLATITUDE;

    // get distance on X axis
    double xDistance = (cos(peerLatitude * (M_PI/180.0)) +
cos(myLatitude * (M_PI/180.0))) * abs(myLongitude - peerLongitude) *
(NAUTICALMILEPERLONGITUDE / 2);

    // find the diagonal distance between two points
    double distance = sqrt(pow(yDistance,2) + pow(xDistance,2));

    // convert the nautical miles into KM
    return int((distance * KMPERNAUTICALMILE) + .5);
}
```

Appendix C

Experimental results

```
mysql> select * from rtt_table where infohash = "0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e" order by rtt;
+----------------+-------+---------+------------------------------------------+
| ipaddr         | port  | rtt     | infohash                                 |
+----------------+-------+---------+------------------------------------------+
| 188.[redacted].126 |  2025 | 30.5411 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 193.[redacted].54  |  1502 | 59.6243 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 88.[redacted].227  | 19534 | 69.8587 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 120.[redacted].28  |  1500 |  174.05 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 59.[redacted].76   |  1267 | 191.049 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 182.[redacted].56  | 13655 | 195.776 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 117.[redacted].255 |  2026 | 221.993 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 223.[redacted].144 |  1505 | 241.878 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 124.[redacted].4   |  6881 | 265.265 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 59.[redacted].38   |  1501 | 336.831 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
+----------------+-------+---------+------------------------------------------+
10 rows in set (0.00 sec)
```

Figure C.1 Table that stores RTT values

```
mysql> select * from ashop_table where infohash = "0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e" order by hopcount;
+----------------+-------+----------+------------------------------------------+
| ipaddr         | port  | hopcount | infohash                                 |
+----------------+-------+----------+------------------------------------------+
| 59.[redacted].38   |  1501 |        2 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 88.[redacted].227  | 19534 |        2 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 188.[redacted].126 |  2025 |        2 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 117.[redacted].255 |  2026 |        2 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 223.[redacted].144 |  1505 |        2 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 193.[redacted].54  |  1502 |        3 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 120.[redacted].28  |  1500 |        3 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 182.[redacted].166 |  1268 |        3 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 59.[redacted].76   |  1267 |        3 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 124.[redacted].4   |  6881 |        3 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 219.[redacted].135 |  1502 |        3 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
| 182.[redacted].56  | 13655 |        3 | 0c1a100e92cf2649ac7a0a6875a48ee7c8bf551e |
+----------------+-------+----------+------------------------------------------+
12 rows in set (0.00 sec)
```

Figure C.2 Table that stores AS-Hop count values

Appendix D

Acronyms

API	Application Programming Interface
AS	Autonomous System
BNS	Biased Neighbor Selection
BU	Biased Unchoking
BGP	Border Gateway Protocol
CDN	Content Delivery Network
CPU	Central Processing Unit
DHT	Distributed Hash Table
DNS	Domain Name System
GTK+	Gimp ToolKit
HTTP	Hypertext Transfer Protocol
ISP	Internet Service Provider
NAT	Network Address Translation
NCS	Network Coordinate System
PDF	Portable Document Format
PEX	Peer Exchange
PNG	Portable Network Graphics
QoS	Quality of Service
RTO	Retransmission Timeout
RTT	Round Trip Time
RTP	Real-time Transport Protocol
SQL	Structured Query Language

TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UML	Unified Markup Language

Appendix E

Glossary

Autonomous System: One or more networks and routers under single administrative control.

AS-Hops: Number of ASes that a specific route passes through to reach the destination

Content Delivery Network: It is a system of distributed servers that stores copies of data. When a user request for data, servers closets to it, respond with the cached content.

Domain Name Systems: It is a naming system for computers or other resources connected to internet. It can be called as an internet service which translates domain name to IP address.

Pymdht: It is the flexible implementation of mainline DHT. The Implementation is in python and is developed at KTH, TSLab.

Peer: A member in a peer-to-peer system.

Score Metric: It corresponds to the geographic distance in kilometer.

Appendix F

Implementation – Protected methods

Name	calculateDistance
Input Parameter	double peerLatitude : Latitude of peer double peerLongitude : Longitude of peer double myLatitude : Our peer Latitude double myLongitude : Our peer Longitude
Output Parameter	double distance : Distance between our peer and Other peer
Description	calculate Geographic Distance

Table F.1 Core internal method - Calculate Distance

Name	calculateRtt
Input Parameter	char* ipAddress : Peer IP address
Output Parameter	double rtt: RTT between our peer and Other peer
Description	Calculate RTT to particular IP address from our peer

Table F.2 Core internal method - Calculate RTT

Name	calculateHopCount
Input Parameter	char* ipAddress1 : Peer IP address char* ipAddress2 : Our peer IP address
Output Parameter	int HopCount: Hop count between our peer and Other peer
Description	calculate AS hop count between two IP addresses

Table F.3 Core internal method - Calculate AS Hop Count

Name	getRecords
Input Parameter	char* ipAddress : Peer IP address
Output Parameter	string& continent : Name of continent string& country : Name of country string& city : Name of city
Description	Get city, country and continent of particular IP address

Table F.4 Core internal method - Find Geo-Location information

Name	isInSameASN
Input Parameter	char* ipAddress : Peer IP address
Output Parameter	bool flag : If peer are from same ASN
Description	Is our peer and other peer are from same ASN

Table F.5 Core internal method - Check if from same ASN

Name	isInSameCity
Input Parameter	char* ipAddress : Peer IP address
Output Parameter	bool flag : If peer are from same city
Description	Is our peer and other peer are from same city

Table F.6 Core internal method - Check if from same City

Name	isInSameCountry
Input Parameter	char* ipAddress : Peer IP address
Output Parameter	bool flag : If peer are from same country
Description	Is our peer and other peer are from same country

Table F.7 Core internal method - Check if from same Country

Name	isInSameContinent
Input Parameter	char* ipAddress : Peer IP address
Output Parameter	bool flag : If peer are from same continent
Description	Is our peer and other peer are from same continent

Table F.8 Core internal method - Check if from same Continent

Name	getASN
Input Parameter	const char* ipAddress : Peer IP address
Output Parameter	int AS number : ASN corresponding to IP address
Description	This API return the ASN of particular IP address

Table F.9 Core internal method - Get AS number from IP address

Name	calculateScore
Input Parameter	char* ipAddress : Peer IP address

Output Parameter	int score : score
Description	Calculate the weighted distance between our peer and other Peer

Table F.10 Core internal method - Calculated weighted distance

Name	initialize
Input Parameter	void
Output Parameter	void
Description	Download MAXMIND database on your local system

Table F.11 Core internal method - initialize

Name	getCoordinates
Input Parameter	char* ipAddress : Peer IP address
Output Parameter	double &latitude : Latitude value double & longitude : Longitude value
Description	get logitude and latitude

Table F.12 Core internal method - Get Coordinates

Name	sortIP
Input Parameter	ipAddress_s ipAddressList[]: Input list int length : Number of peers
Output Parameter	double &latitude : Latitude value double & longitude : Longitude value
Description	Sort peers based on score

Table F.13 Core internal method - Sort IP

Name	sortRTT
Input Parameter	rtt_s rttList[]: Input list int length : Number of peers
Output Parameter	rtt_s rttList[]: Sorted list
Description	sort peers based on RTT

Table F.14 Core internal method - Sort RTT

Name	sortAsHop
Input Parameter	asHop_s ashopList[]: Input list int length : Number of peers

Output Parameter	asHop_s ashopList[]: Sorted list
Description	sort peers based on AS-HOP count

Table F.15 Core internal method - Sort AsHop

Name	getIpAddress
Input Parameter	string hash : infohash/roothash of file
Output Parameter	std::vector<ipPort_s> &list: List of IP addresses associated with a particular file
Description	Fetch list of IP addresses for a infohash from database

Table F.16 Core internal method - Get IP addresses associated with given infohash

Name	storePeerInScoreTable/storePeerInRttTable/storePeerInAsHopTable
Input Parameter	ipAddress_s ipAddressList[] / rtt_s rttList[] / asHop_s asHopList[] : List of peers int length : number of peers string hash : infohash/roothash of file
Output Parameter	void
Description	getpeers method call sortIP/sortRTT etc methods and sort the IP addresses fetch using getIpAddresses, These sorted entries are stored in score_table/rtt_table/ashop_table

Table F.17 Core internal method - Store peers in corresponding table

Name	addPeersInDB / addPeerInDB
Input Parameter	std::vector<ipPort_s> ipPortList : List of peers / string ip, int port : String IP address and port string hash : infohash/roothash of file
Output Parameter	void
Description	These are threads methods are registered in pthread_create APIs. Peers are added using addpeer/addpeers and then they are sorted in seperate thread. After sorting they are stored in database

Table F.18 Core internal method - Create separate thread for computing information

Name	entry / entryAddPeer
Input Parameter	void *arg

Output Parameter	static void*
Description	object of BitSwiftSelector(“this” pointer) is passed to these entry functions. using “this” pointer addPeersInDB/addPeerInDB member functions are called

Table F.19 Core internal method - Entry function

Compilation and linking

Commands
g++ -c -I/usr/include/mysql++ -I/usr/include/mysql -I/usr/local/include/mysql++ -g BitSwiftSelector.cpp g++ -c -I/usr/include/mysql++ -I/usr/include/mysql -I/usr/local/include/mysql++ -g UnitTest.cpp g++ -o BitSwiftSelector BitSwiftSelector.o UnitTest.o -L/usr/local/lib -lmysqlpp -lmysqlclient -lGeoIP -lcurl -lnsl -lz -lm

Table F.20 Compilation and linking commands

Commands
g++ -c -I/usr/include/mysql++ -I/usr/include/mysql -I/usr/local/include/mysql++ -g client_test.cpp g++ -o client_test client_test.o -L/usr/local/lib -lmysqlpp -lmysqlclient -lGeoIP -lcurl -lnsl -lz -lm -ltorrent-rasterbar

Table F.21 Compile client_test and link it with libtorrent-rasterbar

References

[1] S. Oechsner, F. Lehrieder, T. Hoßfeld, F. Metzger, D. Staehle, and K. Pussep, "Pushing the performance of biased neighbor selection through biased unchoking," in *IEEE Peer-to-Peer Computing*, pp. 301–310, 2009.

[2] "*Skype*". [Online]. Available: http://www.skype.com/intl/en/home/. [Accessed: 27-May-2012].

[3] "User Manual - Help - BitTorrent - Delivering the World's Content." [Online]. Available: http://www.bittorrent.com/help/manual/. [Accessed: 28-May-2012].

[4] S. Agarwal and J. R. Lorch, "Matchmaking for online games and other latency-sensitive P2P systems" in *Proceedings of the ACM SIGCOMM conference on Data communication*, vol. 39, no. 4, pp. 315–326, 2009.

[5] B. Liu, Y. Cui, Y. Lu, and Y. Xue, "Locality-awareness in BitTorrent-like P2P applications," *IEEE Transactions on Multimedia*, vol. 11, no. 3, pp. 361–371, 2009.

[6] F. Lehrieder, S. Oechsner, T. Ho\ssfeld, Z. Despotovic, W. Kellerer, and M. Michel, "Can p2p-users benefit from locality-awareness?," in *IEEE Peer-to-Peer Computing*, pp. 1–9, 2010.

[7] Y. Liu, L. Xiao, X. Liu, L. M. Ni, and X. Zhang, "Location awareness in unstructured peer-to-peer systems,", *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 2, pp. 163–174, 2005.

[8] S. Sen and J. Wang, "Analyzing Peer-To-Peer Traffic Across Large Networks," *IEEE/ACM Transactions on Networking*, vol. 12, no. 2, pp. 219–232, Apr. 2004.

[9] A. A. M. Saleh and J. M. Simmons, "Technology and architecture to enable the explosive growth of the internet," *IEEE Communications Magazine*, vol. 49, no. 1, pp. 126–132, 2011.

[10] S. C. Hong, J. Kim, B. Park, Y. J. Won, and J. W. Hong, "Traffic growth analysis over three years in enterprise networks," in *Proceedings of APCC 15th Asia-Pacific Conference on Communications*, pp. 896–899, 2009.

[11] R. Jimenez, F. Osmani, and B. Knutsson, "A flexible implementation of the Mainline DHT protocol," [Online]. Available: https://github.com/rauljim/pymdht. [Accessed: 28-May-2012].

[12] B. Cohen, "The BitTorrent Protocol Specification" [Online]. Available: http://www.bittorrent.org/beps/bep_0003.html. [Accessed: 28-May-2012].

[13] A. Norberg, “The C++ Implementation of BitTorrent Protocol.” [Online]. Available: http://www.rasterbar.com/products/libtorrent/. [Accessed: 28-May-2012].

[14] V. Aggarwal, A. Feldmann, and C. Scheideler, “Can ISPs and P2P users cooperate for improved performance?,” in *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 3, pp. 29–40, 2007.

[15] H. Schulze and K. Mochalski, “Internet Study 2008/2009,” *IPOQUE Report*, 2009

[16] D. R. Choffnes and F. E. Bustamante, “Taming the torrent: a practical approach to reducing cross-isp traffic in peer-to-peer systems,” in *ACM SIGCOMM Computer Communication Review*, vol. 38, pp. 363–374, 2008.

[17]“Vuze - BitTorrent Client,” [Online]. Available: http://www.vuze.com/. [Accessed: 28-May-2012].

[18] R. Bindal, P. Cao, W. Chan, J. Medval, G. Suwala,T. Bates, and A. Zhang “Improving traffic locality in bittorrent via biased neighbor selection,” in *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, Washington, DC, USA, pp. 66, 2006.

[19] H. Xie, R. Y. Yang, A. Krishnamurthy, Y. G. Liu, and A. Silberschatz. “P4P: Provider portal for applications,” in *ACM SIGCOMM Computer Communication Review*, vol 38, no 4, pp.351–-362, 2008.

[20] V. Grishchenko, “The Generic Multiparty Transport Protocol (swift*),” Internet-draft, Internet Engineering Task Force*, October 2011, [Online].Available: http://tools.ietf.org/id/draft-grishchenko-ppsp-swift-03.txt

[21] Guillem Cabrera Anon,"Joining BitTorrent and swift to improve P2P transfers," Master’s Thesis, *Software Technology department in the Faculty of Electrical Engineering, Mathematics and Computer Science of the Delft University of Technology* (The Netherlands), July 2010

[22] H. Schulze and K. Mochalski, “Internet Study 2008/2009,” *IPOQUE Report*, 2009, [Online].Available: http://www.ipoque.com/sites/default/files/mediafiles/documents/internet-study-2008-2009.pdf [Accessed: 28-May-2012].

[23] B. Cohen, “Incentives build robustness in BitTorrent,” in Proceedings of the Workshop on Economics of Peer-to-Peer Systems (P2PEcon), 2003.

[24]“Comparison of BitTorrent clients,” [Online]. Available: http://en.wikipedia.org/wiki/Comparison_of_BitTorrent_clients. [Accessed: 28-May-2012].

[25]A. Loewenstern, “Mainline DHT Specication” [Online]. Available: http://www.bittorrent.org/beps/bep_0005.html. [Accessed: 28-May-2012].

[26] Rüdiger Schollmeier, "A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications," *in Proceedings of the First International Conference on Peer-to-Peer Computing, IEEE*,2002.

[27] "Object Management Group - UML." [Online]. Available: http://www.uml.org/. [Accessed: 28-May-2012].

[28] "Introduction to OMG UML." [Online]. Available: http://www.omg.org/gettingstarted/what_is_uml.htm. [Accessed: 28-May-2012].

[29] "Dia - Homepage," [Online]. Available: https://live.gnome.org/Dia. [Accessed: 28-May-2012].

[30] N. Wahlén, "Autonomous System Distances Package, se.sics.asdistances," [Online]. Available: http://projects.wahni.se/asdistances-api/. [Accessed: 28-May-2012].

[31] M. Balaban, "Introduction to Mysql++ (c++ API for Mysql) on FreeBSD." [Online]. Available: http://www.enderunix.org/docs/en/mysqlcpp.html. [Accessed: 28-May-2012].

[32]K. Atkinson, "MySQL++ Documentation." [Online]. Available: http://tangentsoft.net/mysql++/doc/. [Accessed: 28-May-2012].

[33] "Fundamental Specifications of Kyoto Cabinet Version 1." [Online]. Available: http://fallabs.com/kyotocabinet/spex.html. [Accessed: 28-May-2012].

[34] "MaxMind – GeoIP, IP Address Location Technology." [Online]. Available: http://www.maxmind.com/app/ip-location. [Accessed: 28-May-2012].

[35] "Visio 2010," [Online]. Available: http://visio.microsoft.com/en-us/TryBuy/TryVisio2010forFree/Pages/Try_Visio_2010_for_Free.aspx. [Accessed: 28-May-2012].

[36] Rakesh Kumar, "PeerSelector: Grouping peers in a P2P system - Implementation," [Online]. Available: https://github.com/RakeshKmr/PeerSelector. [Accessed: 28-May-2012].

[37] James F. Kurose and Keith W. Ross, Computer Networking: A Top-down Approach Featuring the Internet, 5th Edition, Addison-Wesley, 2009

[38] M. Piatek, H. V. Madhyastha, J. P. John, A. Krishnamurthy, and T. Anderson, "Pitfalls for ISP-friendly P2P design," in *Proceedings of HotNets-VIII*, 2009

[39] R. Cuevas Rumin, N. Laoutaris, X. Yang, G. Siganos, and P. Rodriguez, "Deep diving into BitTorrent locality," in *Proceedings of IEEE INFOCOM*, pp. 963–971, 2011.

[40] H. Zhang, Z. Shao, M. Chen, and K. Ramchandran, "Optimal neighbor selection in BitTorrent-like peer-to-peer networks," in *Proceedings of the ACM SIGMETRICS joint*

international conference on Measurement and modeling of computer systems, pp. 141–142, 2011.

[41] J. P. Fernandez-Palacios Gimenez, M. A. Callejo Rodriguez, H. Hasan, T. Hoßfeld, D. Staehle, Z. Despotovic,W. Kellerer, K. Pussep, I. Papali, G. D. Stamoulis,and B. Stiller, "A New Approach for Managing Traffic of Overlay Applications of the SmoothIT Project," in *2nd International Conference on Autonomous Infrastructure, Management and Security*, Bremen, Germany, July 2008.

[42] S. Le Blond, A. Legout, and W. Dabbous, "Pushing bittorrent locality to the limit," *Computer Networks*, vol. 55, no. 3, pp. 541–557, 2011.

[43] G. Shen, Y. Wang, Y.Xiong, B.Y. Zhao, and Z.L. Zhang,"Relieving the tension between ISPs and P2P." in *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2007.

[44] "MaxMind - GeoIP City Accuracy for Selected Countries." [Online]. Available: http://www.maxmind.com/app/city_accuracy. [Accessed: 28-May-2012].

[45] "Route Views Project," *University of Oregon*. [Online]. Available: http://www.routeviews.org/. [Accessed: 28-May-2012].

[46] "Peer-to-peer file sharing," [Online]. Available: http://en.wikipedia.org/wiki/Peer-to-peer_file_sharing. [Accessed: 28-May-2012].

[47] T. Klingberg, "Gnutella v0.6 RFC," [Online]. Available: http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html. [Accessed: 28-May-2012].

[48] A. Bakker, V. Grishchenko, R. Petrocco, J. Pouwelse "A Swift Multitasking Transport Protocol as PPSP" [Online]. Available: http://www.ietf.org/proceedings/82/slides/ppsp-1.pdf [Accessed: 28-May-2012].

[49] "Current IP Check." [Online]. Available: http://checkip.dyndns.com/. [Accessed: 28-May-2012].

[50] "MaxMind - GeoIP C API." [Online]. Available: http://www.maxmind.com/app/c. [Accessed: 28-May-2012].

[51] S. Saroiu, P. K. Gummadi, S. D. Gribble, and others, "A measurement study of peer-to-peer file sharing systems," in *proceedings of Multimedia Computing and Networking*, p. 152, 2002.

[52] J. Liang, R. Kumar, and K. W. Ross, "Understanding kazaa," *Manuscript, Polytechnic Univ*, 2004.

[53] “µTorrent - BitTorrent Client,” [Online]. Available: http://www.utorrent.com/ [Accessed: 28-May-2012].

[54] “PlanetLab,” [Online]. Available: http://www.planet-lab.org/ [Accessed: 28-May-2012].