Design and evaluation of a green BitTorrent for energy-efficient content distribution

Jeremy H. Blackburn
University of South Florida
Design and Evaluation of a Green BitTorrent for Energy-Efficient Content Distribution

by

Jeremy H. Blackburn

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science & Engineering
College of Engineering
University of South Florida

Major Professor: Ken Christensen, Ph.D.
Adriana Iamnitchi, Ph.D.
Abraham Kandel, Ph.D.

Date of Approval:
April 6, 2010

Keywords: P2P, peer-to-peer, data center, set-top-box, power management

© Copyright 2010, Jeremy H. Blackburn
Dedication

This thesis is dedicated to my close friends and family who continually inspire and encourage me to reach my potential. In particular, I’d like to thank Zach King and Daniel Nishijima for refusing to let me quit when things looked dark.

To my parents, Bob and Janet, I’d like to thank you for always pushing me to achieve my potential and never giving up on me. Without your unconditional love and guidance I could not have made it to this point.

Finally, to my wife, Lauren. You never let me give in to my doubts, never let me question my ability, and above all have served as my motivation to not only ask questions, but to search for answers.
Acknowledgments

I would like to thank Dr. Ken Christensen at the University of South Florida for providing me the opportunity to write this thesis. For several years Dr. Christensen served as a sounding board and, motivator, and more importantly a mentor. He helped me progress through an Undergraduate Research position, graduate level research, publication and, now to a defending Masters student.

I would also like to thank Dr. Miguel Labrador and the entire University of South Florida Research Experience for Undergraduates program. This thesis began as a project from the REU program and the encouragement of all individuals involved facilitated my transition from undergraduate research to graduate research.

Many thanks to Drs. Iamnitchi and Kandel for both being members of my committee as well as mentors in the class room, both for undergraduate and graduate studies.

The development of this material is based upon work supported by the National Science Foundation under grant 0754537 (Blackburn) and CNS-0520081 (Christensen). I would also like to acknowledge the use of high performance computational services provided by Research Computing, University of South Florida.

Finally, I would like to thank my brother, Dr. Bryan Blackburn of the University of Maryland for providing me with a unique view of academic research and providing regular reality checks. Without his example and our relationship which allowed for honest and frank meta-discussions of academia, this thesis would not have been possible.
# Table of Contents

List of Tables iii
List of Figures iv

Abstract vi

Chapter 1: Introduction 1
1.1 Data Centers for Content Distribution 1
1.2 Content Distribution Networks 2
1.3 Peer-to-Peer for Content Distribution 3
1.4 Energy Use of Content Distribution 4
    1.4.1 Data Centers 5
    1.4.2 Peer-to-Peer 5
1.5 Motivation 5
    1.5.1 Opportunity for Savings 7
1.6 Contribution 7
1.7 Organization of this Thesis 8

Chapter 2: Background and Literature Review 9
2.1 Client-Server Content Distribution 9
2.2 Peer-to-Peer Content Distribution 10
2.3 The ns-2 BitTorrent Simulator 14
2.4 The BitTornado BitTorrent Client 15
2.5 Related Work 21
2.6 Chapter Summary 22

Chapter 3: Green BitTorrent Protocol Extension 23
3.1 What is a Green BitTorrent? 23
3.2 New Peer States 23
3.3 New Peer Events 24
3.4 Backwards Compatibility 26
3.5 Chapter Summary 26

Chapter 4: Implementation of Green BitTorrent 27
4.1 Changes to ns-2 BitTorrent Simulation 27
    4.1.1 Peer States 27
    4.1.2 Peer Events 27
4.2 Changes to BitTornado 29
    4.2.1 Peer States 29
    4.2.2 Peer Events 30
4.3 Chapter Summary 31
List of Tables

Table 1. Power consumption of various devices 6
Table 2. BitTorrent protocol messages 13
Table 3. Increase in download time for Green BitTorrent versus standard BitTorrent (simulation) 38
Table 4. Increase in download time for Green BitTorrent compared to standard BitTorrent 44
Table 5. Download time of real client versus simulation client 47
List of Figures

Figure 1. Data center model for content distribution 2
Figure 2. Content Distribution Network model for content distribution 3
Figure 3. P2P model for content distribution 4
Figure 4. High level view of a BitTorrent swarm 11
Figure 5. Torrent class diagram 15
Figure 6. Downloader class diagram 16
Figure 7. SingleDownload class diagram 17
Figure 8. Uploader class diagram 19
Figure 9. Encoder class diagram 20
Figure 10. Connector class diagram 20
Figure 11. Description of changed peer disconnect detection event 24
Figure 12. BitTornado peer disconnect event [12] 24
Figure 13. Description of changed peer discovery event 25
Figure 14. Description of new peer inactivity and sleep event 25
Figure 15. Description of new peer wake up event 26
Figure 16. Class hierarchy diagram for Green BitTorrent implementation in BitTornado 29
Figure 17. GreenConnector#connection_lost() method 30
Figure 18. GreenEncoder#start_connection() method 30
Figure 19. GreenConnector#connection_made() method 31
Figure 20. GreenTorrent#try_to_sleep() and GreenTorrent#go_to_sleep() methods 31
Figure 21. GreenTorrent#wake_up() method 31
Figure 22. Download time for standard BitTorrent (simulation) 36
Figure 23. Sleep time for standard BitTorrent (simulation) 36
Figure 24. Sleep time for peer 25 standard BitTorrent (simulation) 37
Design and Evaluation of a Green BitTorrent for Energy-Efficient Content Distribution

Jeremy H. Blackburn

ABSTRACT

IT equipment has been estimated to be responsible for 2% of global CO₂ emissions and data centers are responsible for 1.2% of U.S. energy consumption. With the large quantity of high quality digital content available on the Internet the energy demands and environmental impact of the data centers must be addressed. The use of peer-to-peer technologies, such as BitTorrent, to distribute legal content to consumers is actively being explored as a means of reducing both file download times and the energy consumption of data centers. This approach pushes the energy use out of the data centers and into the homes of content consumers (who are also then content distributors). The current BitTorrent protocol requires that clients must be fully powered-on to be participating members in a swarm.

In this thesis, an extension to the BitTorrent protocol that utilizes long-lived knowledge of sleeping peers to enable clients to sleep when not actively distributing content yet remain responsive swarm members is developed. New peer states and events required for the protocol extension, the implementation the new protocol in a simulation environment, and the implementation of the protocol extension in a real client are described.

Experiments on a simulated swarm of 51 peers transferring a 1 GB and a real swarm of 11 peers transferring a 100 MB file were run. To validate the simulation a simulated swarm of 11 peers transferring a 100 MB file is compared to the real swarm of 11 peers. The results of standard BitTorrent are compared to the new Green BitTorrent by examining download times, sleep time, and awake time. The results of the experiment show significant energy savings are possible with only a small penalty in download time. Energy savings of up to 75% are shown with download time increases as little as 10%. These energy savings could equate to over $1 billion dollars per year in the US alone if Green BitTorrent is used instead of standard BitTorrent for future rollouts of legal distribution systems.
Chapter 1:
Introduction

Maintaining a sustainable society is one of the more pressing challenges facing our modern society. While the Internet revolution has generally been a positive force in the world, until recently the impact on the environment has been overlooked. IT equipment is a large consumer of energy and the green house gasses generated have been estimated to account for 2% of global emissions [20]. As more and more content is delivered digitally, so too will the energy consumed by the distribution, storage, and generation of the content increase. This thesis explores a solution to digital content distribution that addresses and attempts to minimize the energy consumed. By decoupling application state from network connection state in BitTorrent, significant energy savings can be achieved with minimal increase in download time.

An estimated 1.2% of U.S. electricity consumption in 2006 was directly attributable to data centers [29]. This energy usage corresponds to approximately $4.5 billion (at commercial energy rates) [39]. While data base and application servers also make use of data centers, content distribution is quickly becoming a heavy load. For example, the iTunes Store is currently the top music retailer in the US, having sold over four billion songs from April 2003 to April 2008 [6]. As the entirety of the iTunes Store catalogue is digital content, an extremely conservative estimate of 3 MB per song translates to over 11 PB of data transferred over a 5 year period, or 2.2 PB per year in the US alone. While digital music was the sole original product sold via the iTunes Store, as of April 2007, the iTunes Store has sold over 50 million TV shows and two million movies [7].

1.1 Data Centers for Content Distribution

The iTunes Store distributes content via a large centralized architecture [33]. This centralized architecture is referred to as a data center and is currently the prevalent method of content distribution. A data center is a large collection of computers operated by a single entity most often housed in the same physical location acting as a single location from which content is downloaded. All the content available is stored at this single location and any number of content consumers will access the data center to download their content at any given time. Figure 1 shows a data center distribution model. Of note is the single distribution entity (the data
center) responsible for distribution of content with each of the data consumers acting only as receivers of content.

In general, data centers make use of economy of scale for both hardware costs and associated costs (e.g., cooling). Of great interest is that the power and cooling costs of data centers surpass the costs of the equipment itself [11]. This has lead to a search for an alternative to data centers for content distribution.

1.2 Content Distribution Networks

While data centers are a collection of computers at a single location, a Content Distribution Network (CDN) is a logical grouping of computers that may or may not be located together. The usage of CDNs is motivated by the need for high availability and low access times to content. Because of variations in geographic location, cross network latency, and the number of people accessing content at any given time, caching is often used. A caching system replicates content on numerous servers with the goal of being able to provide a copy of the content to a user from a geographically close location. The result from caching is reduced download times (from the perspective of the user), less redundant data transferred over the network, and a reduced load on servers overall [30].

Akamai [2] is the largest commercial CDN provider. The Akamai CDN operates by “Akamaizing” URLs to provide contextual information to Akamai’s servers. An Akamaized URL has an Akamai domain prepended to the original URL as well as metadata which is used to determine which server in the CDN
should respond to the request and whether or not information in the CDNs cache should be updated. Akamai’s products deal with a wide range of content delivery including static and streaming media as well as web applications.

When a user makes a request for content the CDN locates an appropriately close CDN server. The definition of “close” is somewhat malleable as numerous measures for closeness can be used. The primary measures of closeness are geographical location, topological, and network latency [30]. In Figure 2 homes are served by the data center in the CDN that is “closest” to them. While CDNs make a best effort to server users with content via a low latency connection in practice the sub-optimal servers are often chosen. However, while in a small number of cases users would be better served by origin (i.e., non CDN) servers, CDNs have been shown to have a measurable impact on performance by serving content from reasonably “good” servers in the general case [28].

1.3 Peer-to-Peer for Content Distribution

An emerging alternative to data centers for content distribution is peer-to-peer (P2P) technologies. The defining characteristic of P2P distribution technologies is that content consumers do not receive content from
Figure 3. P2P model for content distribution

A centralized location but rather from other content consumers. In a P2P distribution scheme there is no clear delineation between content distributor and content consumer. As seen in Figure 3 each content consumer not only receives content but also distributes it, reducing the number of servers housed in data centers necessary to distribute content.

Some estimates attribute between 18% and 35% of all Internet traffic to BitTorrent [35] making it the dominant P2P distribution protocol in the world today. The success of P2P in general and BitTorrent in particular has not been lost on content distributors [38]. As P2P blurs the line between consumer and distributor it effectively removes the cost of content distribution out of the data center and into the consumers home.

BitTorrent makes even more sense as the quantity of digital content increases. In particular, the problem of flash crowds are prevalent in data centers. A flash crowd occurs when some particular content becomes popular very quickly resulting in a sudden increase in the number of downloads over a short period of time [17]. If the effects of a flash crowd are underestimated load on the data center servers can become very high and download times will increase substantially. This scaling problem, however, does not exist in BitTorrent. As each BitTorrent peer involved in the transfer of content contributes resources in addition to consuming them the distribution network becomes stronger as more peers participate in a flash crowd.

1.4 Energy Use of Content Distribution

About 2% of the global CO₂ footprint has been attributed to the IT industry [20]. This is approximately the same CO₂ output of the aviation industry [20]. The 10 million computers housed at data centers consume
an estimated 61 TWh of electricity per year [29]. These numbers give rise to the desire to reduce the growing energy consumption of the IT industry.

1.4.1 Data Centers

Content distribution in particular is an interesting problem for data centers. Due to the sporadic nature of “on demand” content consumption habits, data centers may very well have low utilization levels [31]. For example, in order to have the bandwidth and computing power to serve the needs of a new digital movie release it is necessary to estimate the peak demand for the movie. This estimation of resources, or dimensioning, results in the resources provisioned for a data center. Too low of an estimation will leave content consumers frustrated with long download times while too high of an estimation will result in wasted resources. Important to note however is that this estimation must be a peak demand estimation. While it is true that at some point in time the demand might be high, there will be numerous other times that the demand will be much lower. Due to the difficulties of dimensioning, servers in data centers generally run at 30% utilization [15]. The low utilization of servers in a data center is important because a computer is less energy efficient the lower its utilization is [9]. Even an energy efficient server is consuming about half of its peak power consumption at 50% utilization.

1.4.2 Peer-to-Peer

P2P makes sense with respect to the provisioning and utilization levels of data centers, however the issue is not entirely unequivocal. As of 2006 there are an estimated 102.5 million home broadband users, or about 72% penetration [10]. If content distributors follow through on plans to move away from data centers and towards P2P distribution, the 22 TWh/yr of power used is comparable to the direct power consumption of computers in data centers per year (20 to 30 TWh) [32]. As home broadband penetration increases so too will the costs placed on consumers.

1.5 Motivation

Fitting with content distributors desire to move distribution out of the data center and into the consumer’s home, a future where dedicated file sharing devices, either as a stand alone unit on the side of a house or as a component of set top boxes, becoming the primary method of content distribution is foreseen. This device could be owned by the operator and would provide for on demand content delivery to the customer as well as a means for operator owned content to be distributed to other customers. Commercial offerings such as the
<table>
<thead>
<tr>
<th>Device</th>
<th>Power Consumption (watts)</th>
<th>Annual Energy Consumption (kWh/yr)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volume Server [29]</td>
<td>183</td>
<td>1,600</td>
</tr>
<tr>
<td>Mid-range Server [29]</td>
<td>423</td>
<td>3,705</td>
</tr>
<tr>
<td>High-end server [29]</td>
<td>4874</td>
<td>42,696</td>
</tr>
<tr>
<td>Desktop PC</td>
<td>100</td>
<td>876</td>
</tr>
<tr>
<td>HD Receiver with DVR [23]</td>
<td>35</td>
<td>350</td>
</tr>
<tr>
<td>Stand-alone DVR [23]</td>
<td>23</td>
<td>200</td>
</tr>
<tr>
<td>Media receiver box [23]</td>
<td>4</td>
<td>35</td>
</tr>
<tr>
<td>P2P file sharing device</td>
<td>25</td>
<td>219</td>
</tr>
</tbody>
</table>

VUDU on-demand movie service [43] are already available to consumers allowing access to digital content via a P2P network and runs on 24 W [43]. These power consumption of these file sharing devices would be much less than a typical PC. The power consumption of servers, PCs, set-top-boxes, and a possible P2P box can be seen in Table 1. At 100 million homes and 25 W per file sharing device a P2P content distribution system results in approximately 22 TWh and $2.2 billion per year (at consumer electricity rates of $0.10 per KWh) of energy use providing the impetus for investigation into methods for increasing the energy efficiency of P2P.

Current P2P systems make the assumption that peers participating in the distribution of content are “always on.” In other words, if a peer is involved in distributing content it will be fully powered on and available for network communications. Unfortunately this assumption precludes putting peers not currently needed to sleep as if they are needed in the future they will not be available and thus deemed not interested in distribution content. In fact, peers going to sleep to save energy would have disastrous consequences in current P2P distribution schemes as the overlay network would breakdown from the lack of available peers. Key to the removal of a centralized download location is that content is replicated across numerous peers in a distributed fashion. That is, peers must be accessible in an on demand fashion to distribute content. P2P content distribution makes provisions for an environment where peers may become unavailable, however the assumption remains that if a peer is involved in distributing content it will remain available and an peer’s unavailability indicates a peer will not participate in distributing content. This distinction between interest and disinterest in distribution is what leads to the demand for a fully powered up peer. Peers cannot simply sleep when they are not actively distributing content as the ability to establish a network connection is necessary to participate in content distribution. At minimum, prior to any data transfer occuring peers must initiate a TCP connection to communicate over. A sleeping peer is unreachable and thus no TCP connection to it can be initialized.
Further, the TCP connections of a peer that transitions to a sleep state will not be maintained and thus all channels of communication to other peers will be closed.

Additionally there is very little computation that is involved in content distribution. While there are computational tasks involved such as hashes to check the integrity of transferred data and possibly encryption for security or Digital Rights Management purposes, the vast majority of a P2P content distribution process involves data transfer over the network, leaving the processor of the device at relatively low utilization. More importantly, P2P content distribution utilization tends to be sporadic. As in the data center model, downloads are not constantly occurring and in contrast to the data center model, it is very unlikely that any single file sharing device will have a local copy of all content available for distribution further reducing the number of possible downloads at any given time.

### 1.5.1 Opportunity for Savings

Is there any inherent reason that P2P devices must be fully powered on at all times? As previously noted, this requirement is due to the assumption that a peer involved in the distribution of content can be available via the network at any time, and thus fully powered on. Instead, it is proposed that peers in the network be aware of the possibility that the peer being communicated with may be sleeping to save energy. With this new knowledge, peers in a P2P content distribution system can behave in a truly “on demand” fashion; sleeping when resources are unneeded and waking up when they are. In this manner, the energy overhead of data centers is reduced while also reducing the inefficiencies of P2P distribution schemes.

### 1.6 Contribution

By addressing the motivation outlined in the previous section, this research contributes the following:

1. The design of a Green BitTorrent Protocol Extension to the standard BitTorrent protocol that removes the assumption that a peer involved in the distribution of content is fully powered on and awake.

2. The implementation of a real Green BitTorrent Client. By modifying an existing BitTorrent client to make use of the protocol extension both the feasibility and real world energy savings possible are shown.

3. An evaluation of Green BitTorrent that through both real and simulated results shows the energy savings possible as well as the minimal penalties associated with the protocol extension.
1.7 Organization of this Thesis

The remainder of this thesis is organized as follows:

• Chapter 2 contains background and a literature review describing current research into the energy efficiency of content distribution and a review of the BitTorrent protocol.

• Chapter 3 describes the new Green BitTorrent as well as the design of the Green BitTorrent protocol extension.

• Chapter 4 describes the implementation of the Green BitTorrent protocol extension as both a simulation and a real client.

• Chapter 5 describes an evaluation of the Green BitTorrent protocol extension.

• Chapter 6 concludes the thesis with a discussion of the benefits of the contributions and direction for future research.
Chapter 2:
Background and Literature Review

In order to understand how energy can be saved it is first necessary to understand where it is wasted and the constraints that lead to its waste. This chapter discusses the information necessary to understand how the ideas presented in the following chapter result in energy savings.

2.1 Client-Server Content Distribution

By far the most common client-server content distribution system is the World Wide Web. The architecture of the web is the prototypical client-server model: content is stored on a web server and via the HTTP protocol transferred to web browser on users’ machines. The majority of content served via the web is small, but large content is also distributed [3] [21] [36]. While web servers can function as distributors of large content there are several reasons why doing so is not desirable. As discussed previously the problems of over provisioning, the high cost of induced power usage in data centers, and scalability are all issues.

Besides static content for which the download must be fully completed before the content can be used, there also exists streaming content. Streaming content is accessed while it is being transferred. For example, a content consumer might begin watching a movie before the entirety has been downloaded. While only 1% of overall requests for content are of the streaming variety, they represent approximately 20% of the data transferred from CDNs over HTTP [30]. Youtube.com streams well over 100 million videos a day with users uploading 20 hours of new video every minute [44]. While Youtube’s content is user generated, services such as Hulu.com deliver commercially produced streaming TV and movies to content consumers. Streamed content is delivered over numerous protocols and formats, including HTTP, Microsoft Media Services (MMS), Apple Quicktime, and Flash Video. Specialized server applications exist for each of these protocols and formats. Adobe provides two classes of streaming servers: Flash Media Interactive Server which is intended for real time streams and Flash Media Streaming Server for less complex streams. In addition to these streaming formats which are typically delivered to computers via a web browser plugin, a new set of dedicated streaming services has arisen. Netflix Streaming, telecom Video on Demand offerings, and consumer products such as AppleTV all deliver content to consumers in a streaming fashion. But these streaming distribution mecha-
nisms are still inherently client server. Content is placed on a server to which clients connect. When a client connects and requests a stream the server begins sending datagrams which may or may not arrive at the client. Upon receiving a datagram the client decodes it and displays the content.

2.2 Peer-to-Peer Content Distribution

Napster was the first well known P2P system, characterized by its centralized content look-up server which kept track of which peers had which content. A Napster peer looking to download content would first query this centralized server which would return a list of peers the content might be retrieved from. The Napster peer would then choose a single peer from this list and begin download. Gnutella is another popular first generation P2P system. Instead of a centralized look-up server, Gnutella queries are handled in a peer-to-peer fashion.

A swarming P2P system works in a different fashion than first generation P2P systems. While peers in first generation systems both receive and distribute content this behavior is on a one-to-one basis. For any given piece of content a peer will receive it from a single other peer. In a swarming P2P system peers download from multiple peers at a time.

While the cooling costs associated with data centers do not fully disappear with P2P, they are greatly reduced. Simply put, because P2P nodes are not congregated in a central location the existing in place cooling at the location of P2P nodes suffice and no special cooling concerns must be addressed. As cooling accounts for about 50% of the power consumption of data centers it is apparent that P2P systems have at least one benefit. Further, while P2P devices obviously require some cooling it can be argued that this need is met with the ambient cooling provided by air conditioning for use by humans [33].

While the energy consumption of a P2P system has been modeled previously [33], a different assumption is made as to the availability of peers in the system. The assumption is made that P2P clients will run on top of already on machines, and it is explicitly noted that no machines will be kept awake solely to participate in content distribution. The conclusions drawn from this model are that P2P systems are more efficient in terms of the power consumed (directly and indirectly) by the machines participating in content distribution while centralized systems (i.e., data centers) fare better with power consumed by the network.

The BitTorrent protocol has become the dominant swarming P2P protocol in the world. The major difference between BitTorrent and other swarming P2P protocols is the tit-for-tat mechanism that BitTorrent uses to enforce resources sharing. Peers distributing content via BitTorrent act as both clients and servers while the
tit-for-tat mechanism attempts to ensure both an optimal resource distribution as well as manage scalability issues.

While the meaning of “peer” in BitTorrent is essentially the same as in other P2P systems, due to its swarming nature peers are further divided into two distinct classifications: seeds and leeches. Seeds are peers that have a complete copy of the content being distributed while leeches do not yet have a complete copy. Leeches participate in a swarm by both uploading and downloading pieces while seeds only upload. Next discussed are the workings of the BitTorrent protocol and how the distinction between seed and leech makes energy savings possible.

The underlying workings of BitTorrent are as follows. A client seeking to retrieve content acquires a .torrent file through a means outside of BitTorrent (often an HTTP server). The .torrent file itself is the metadata describing the content to be distributed (hereby known as a torrent). A torrent file contains the host name of one or more trackers and a list of pieces (and their checksums) of the torrent. A piece is simply a subdivision of the content described in the torrent. A tracker is a host that contains a list of peers registered as having interest in the distribution of the torrent associated with a particular torrent file. The collection of all peers interested in a torrent across all trackers is known as a swarm.
A high level view of a BitTorrent swarm can be seen in Figure 4. A file is broken up into four pieces. There is one seed that has all the pieces and two leeches; one with two pieces and one with no pieces. The seed is transferring piece 1 to the leech with no pieces as well as piece 3 to leech with two pieces. The leech with two pieces is transferring piece 2 to the leech with no pieces. Again, the ability to upload without having a complete copy of the file being transferred is what makes BitTorrent unique among swarming P2P protocols.

Once a client has downloaded a torrent file and registered its interest with a tracker (making it a peer in the swarm) it retrieves a randomly selected list of up to 50 other hostnames for peers in the swarm. The hostnames for these peers are entered into a peer list and the client initiates a connection (via TCP) to a configurable number (max_connect) of peers in its local peer list. The format of the peer list is implementation specific but at minimum associates a given peer’s relevant information with a unique identifier. If a peer in the peer list cannot be connected to it is ejected from the list. If the size of the peer list falls below a configurable variable additional peers are requested from the tracker. Of note is that connections initiated from other peers might also result in additions to the peer list.

The other peers connected to any given peer are known as that peer’s local swarm. Peers periodically exchange information on what pieces they have with their local swarm. This information exchange allows each peer to request the pieces it is missing from local swarm members that have them.

To ensure piece propagation and maximize download throughput BitTorrent peers employ a choking algorithm where upload bandwidth is reciprocated with download bandwidth. This tit-for-tat mechanism provides upload bandwidth to peers in the local swarm in proportion to the download bandwidth they are providing for the peer. Peers in the local swarm are periodically ordered by their download bandwidth contribution and a limited number are provided reciprocal upload bandwidth (unchoked) and the rest are provided no upload bandwidth (choked). Additionally, for bootstrapping and discovery purposes a choked peer is optimistically unchoked, disregarding the ordering by download bandwidth contribution.

Peers communicate via a set of messages. The standard BitTorrent protocol [16] defines eight messages: CHoke, UNCHoke, INTERESTED, NOT_INTERESTED, HAVE, BITFIELD, REQUEST, PIECE, and CANCEL. The messages can be broken up into three logical groupings: content transfer, discovery, and resource management. Table 2 shows these messages and their associated payloads.

The content transfer grouping of messages includes the REQUEST, PIECE and CANCEL messages. When a peer wishes to receive a piece from another peer it sends a REQUEST message (with the payload indicative of the piece it is requesting). The peer on the receiving end of the REQUEST message responds
Table 2. BitTorrent protocol messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQUEST</td>
<td>an index, begin, and length of the piece being requested</td>
</tr>
<tr>
<td>PIECE</td>
<td>an index, begin, and the piece being sent</td>
</tr>
<tr>
<td>CANCEL</td>
<td>an index, begin, and length of the piece being cancelled</td>
</tr>
<tr>
<td>INTERESTED</td>
<td>(no payload)</td>
</tr>
<tr>
<td>NOT_INTERESTED</td>
<td>(no payload)</td>
</tr>
<tr>
<td>HAVE</td>
<td>index of piece sender just completed</td>
</tr>
<tr>
<td>BITFIELD</td>
<td>bitfield with 1 if piece is available, 0 otherwise</td>
</tr>
<tr>
<td>CHOKE</td>
<td>(no payload)</td>
</tr>
<tr>
<td>UNCHOKE</td>
<td>(no payload)</td>
</tr>
</tbody>
</table>

with a PIECE message (with the payload indicative of the piece it is sending). CANCEL messages are used during the “endgame” mode that occurs towards the end of a download. The endgame mode involves a peer flooding all connected peers with requests for the remaining pieces. Once the peer receives the final pieces it CANCELS any outstanding requests. The discovery grouping of messages includes the INTERESTED, NOT_INTERESTED, HAVE, and BITFIELD messages. A peer sends an INTERESTED message to peers to indicate that it has an interest in the pieces the recipient of the INTERESTED message has available. Conversely, a NOT_INTERESTED message is sent if the recipient has no pieces which the sender does not.

The HAVE message is sent to all peers in a local swarm upon the completed download (and successful hash check) of a piece and is the method that already connected peers learn about newly available pieces in their local swarm. The BITFIELD message is the first message exchanged between peers as part of a protocol level handshake. Its payload is a bitfield indicating which pieces the sender has available, with available indexes set to 1 and other set to 0. Finally, the resource management grouping of messages contains the CHOKE and UNCHOKE messages. A peer sends another peer a CHOKE message indicating that REQUEST messages will no longer be honored and an UNCHOKE message to indicate that they will be honored.

In terms of wasted energy the important items of note are the maintenance of the peer list, and the related need for other peers to be able to initiate connections. While an inability to initiate a connection to a peer results in its ejection from the peer list, the termination of an existing connection also results in ejection. In either case, a non communicating peer is marked as “dead” and is ejected from the peer list. As a sleeping peer is unable to respond to initiated TCP requests, and a peer going to sleep severs the connections to its local swarm, an ejection from the peer list necessarily follows. This forces power management to be disabled for a BitTorrent client to continue contributing pieces after becoming a seed even when lightly utilized (i.e., it has very few upload requests).
2.3 The ns-2 BitTorrent Simulator

The ns-2 network simulator is a C++ based discrete event simulation environment used for network research [34]. Applications, devices, communication protocols, and network topologies can all be modeled in ns-2. Due to its modular nature it is relatively easy to add new components to ns-2.

The potential energy savings of a Green BitTorrent client were evaluated using the ns-2 simulator [34]. The BitTorrent packet-level simulation model developed by Eger et al. [19] was used as a starting point. The model by Eger et al. includes the underlying TCP model that is part of ns-2. The BitTorrent model is modular allowing for replacement of peer and piece selection algorithms. This model was modified to implement the green BitTorrent events described in Chapter 3.

As the model created in [19] was initially developed to analyze packet-level vs flow-level BitTorrent simulations it does not fully implement the BitTorrent protocol. The simulation model does not use a torrent file and instead simulates this aspect of BitTorrent via the simple storage of the amount of data to be transferred and the size of pieces. Pieces are not hashed and checked for integrity upon reception and thus any data received by a simulated client is treated as legitimate. The tracker protocol is not implemented and instead a simple tracker-like object is directly accessed by simulated peers (i.e., peers do not communicate with the tracker over the network). Only the basic choking algorithm from [17] is implemented and thus performance related additions such as end game mode are not present.

The ns-2 BitTorrent simulation model is overall reasonably simple in design. A peer is represented by the BitTorrentApp class. BitTorrentApp implements all the functionality of a BitTorrent client including establishing connections between peers, sending and receiving messages, sending and receiving data, running the choking algorithm, and piece selection. BitTorrentApp objects are connected to each other via the built in TCP connection model of ns-2. An additional application protocol connection, BitTorrentConnection wraps the actual simulated TCP connection. Outbound messages are stored in a FIFO buffer which is then cleared by the destination peer upon reception. The application level messages themselves are represented by the BitTorrentData class and all the message types (and associated payloads) listed in Table 2 are available. The tracker is represented by the BitTorrentTracker class and acts as a combined metadata store (similar to the functionality of the torrent file in a real client) and BitTorrent tracker. Again, the tracker protocol is not implemented and thus peers directly access the tracker object to retrieve peer lists as well as the metadata that would normally be present in a torrent file.
2.4 The BitTornado BitTorrent Client

The BitTornado T-0.3.17 BitTorrent client was chosen to base the real Green BitTorrent client off of [12]. BitTornado is based off the original BitTorrent client, with a similar interface and a few additional features, in particular a bandwidth limiter. BitTornado is written in Python and thus multi-platform.

Due to the modular nature of BitTornado there are several components that while not directly modified by a Green BitTorrent implementation are still affected. Additionally, due to the somewhat loose nature of the BitTorrent protocol specification it is worthwhile examining some of the behavior the specification leaves up to client implementation. As BitTornado was forked from the original BitTorrent client code there were some initial changes to organization made. While the majority of the code base was quite modular and well organized the high level code responsible for the client’s interaction with a torrent was not. To this end a Torrent class with common functionality was abstracted away. The Torrent class is responsible for reading the metainfo from a .torrent file, initializing data blocks, initializing the objects necessary for downloading and uploading data, and is a general point of interaction with the client. A class diagram of important attributes and methods of this class can be seen in Figure 5.

While not exhaustive, the important attributes of the Torrent class seen in Figure 5 are:

- **downloader**
- **uploader**
- **encoder**
- **connector**
The downloader attribute is an object that coordinates data downloads. The uploader attribute is an object that coordinates data uploads. The encoder attribute is an object that coordinates connections between peers.

The connector is an object that implements protocol level communication between peers.

The relevant methods are:

- `init_connector()`
- `init_encoder()`
- `start_torrenting()`
- `shutdown()`
- `get_stats()`

`init_connector()` initializes the connector object this Torrent will be using. `init_encoder()` initializes the encoder object this Torrent will be using. `start_torrenting()` begins the process of participating in a swarm. `shutdown()` disconnects from the swarm and shuts the Torrent down. `get_stats()` retrieves important statistics about the Torrent.

The Downloader class is responsible for coordinating data downloads. The primary tasks for downloading data are the request of pieces from other peers and handling of protocol messages that affect data download. A class diagram of important attributes and methods of this class can be seen in Figure 6.

While the Downloader objects coordinates downloads, a separate SingleDownload object for each connected peer which performs the necessary tasks. A class diagram of important attributes and methods of this class can be seen in Figure 7.
The `SingleDownload` class contains the methods:

- `got_choke()`
- `got_unchoke()`
- `is_choked()`
- `_request_more()`
- `got_piece()`
- `got_have()`
- `got_have_bitfield()`

`got_choke()` is an event handler that responds to a connected peer choking this Torrent. As the BitTorrent protocol specifies that a choked peer will not have its piece requests honored, the `got_choke()` handler clears any queued requests for data from the peer that sent the choke message and notifies other components of any requests that were cleared. Similarly, `got_unchoke()` responds to a connected peer unchoking this Torrent and begins the process of requesting more data. `is_choked()` simply returns whether or not this peer is currently choked by the connected peer.

`_request_more()` is the method that queues requests for data. First, an assertion is made that this Torrent is not choked by the peer from which data is to be requested. Next, assuming that the request queue has
not been exceeded a backlog (which starts at 2 and is continually adjusted based on rate and a configurable maximum for optimal performance) new requests are chosen for the request queue. These new requests are chosen via PiecePicker object. The PiecePicker chooses an appropriate piece to request from the connected peer by first choosing the rarest piece first. That is, the piece requested will be the one that the least number of peers in the local swarm have and that the peer the piece is being requested from has. If a suitable piece is found the requested piece is added to the request queue of this peer and a REQUEST message is sent to the connected peer. If no suitable pieces are found, the connected peer will be sent a NOTинтересЕD message.

The got_piece() method is an event handler fired when this peer receives a piece from the connected peer. First, the requested piece is removed from the request queue. Next, the rate measure to the connected peer is updated for use in the choking algorithm. Once this is completed, the piece is validated by comparing the checksum of the received data against the checksum listed in the metainfo file. If the checksum fails, the piece is returned to the request queue. Finally, a check on whether or not the entire file has been downloaded occurs. If the file is complete the Torrent enters seed mode, if it is not, a call to request_more() is made.

The got_have() method is an event handler fired when this peer receives a HAVE message from the connected peer. The PiecePicker is updated to make note of the new piece available from the connected peer. got_have_bitfield() handles the bitfield that is received during a protocol handshake. It operates in a similar fashion as the got_have() handler but works on multiple pieces at a time.

As the Downloader is responsible for coordinating data downloads, the Uploader is responsible for coordinating data uploads. As data uploads are more independent than downloads there is no need for managing Uploaders between connected peers, so unlike the Downloader each connected peer has a single Uploader associated with it. A class diagram of important attributes and methods of this class can be seen in Figure 8.

Relevant methods within the Uploader class are:

- got_not_interested()
- got_interested()
- is_interested()
- got_request()
- got_cancel()
- choke()
When this peer receives a NOT_INTERESTED message from the connected peer, the `got_not_interested()` event handler is fired. The connected peer is marked as not interested, any outstanding data to upload to the connected peer is cleared, and the choking algorithm is re-run. Upon receiving an INTERESTED message, the connected peer is marked as interested and the choking algorithm is re-run via the `got_interested()` event handler. `is_interested()` is used to query whether or not the connected peer is interested in any pieces this peer has completed.

The `got_request()` event handler is fired when this peer receives a REQUEST message from a connected peer. If the peer that sent the REQUEST message is choked then no action occurs. If the requesting peer is not choked the requested piece is added to an outgoing request buffer to be uploaded. If a CANCEL message is received, the `got_cancel()` event handler removes the specified piece from the outgoing request buffer.

If the choking algorithm chooses the connected peer associated with a particular `Uploader` the `choke()` method is called. It marks the connected peer as choked and sends the CHOKE protocol message. Once the CHOKE message has been sent to the choked peer `choke_sent()` clears the outgoing request buffer. If, on the other hand, the choking algorithm chooses the connected peer to be unchoked the `unchoke()` method is called. If the connected peer is currently unchoked no action occurs. If the connected peer is currently choked, however, it is marked as unchoked and sent an UNCHOKE protocol message. `is_choked()` is used to query whether or not the connected peer has been choked by this peer.
Encoder objects manage connections between peers. The connections managed by the Encoder are a BitTorrent protocol abstraction wrapping a TCP/IP socket. A class diagram of important attributes and methods of this class can be seen in Figure 9.

Relevant methods in the Encoder class are:

- `send_keepalives()` - Sends BitTorrent keep alive messages to all connected peers per the BitTorrent protocol specification.
- `start_connection()` - Creates a BitTorrent connection to a peer

The Connecter [sic] class implements BitTorrent protocol level logic. A class diagram of important attributes and methods of this class can be seen in Figure 10.

The Connecter class has the following methods:

- `connection_made()`
- `connection_lost()`
- `got_message()`
- `got_piece()`

The `connection_made()` and `connection_lost()` methods are called when a connection to another peer is made or lost, respectively. When a connection is made it is associated with a new Uploader object. Additionally it is associated with a new SingleDownload object via the Downloader object associated with this
Torrent. Finally, the choker is notified of the connection so the newly connected peer can be included in the choking algorithm.

`got_message()` handles reception of the BitTorrent protocol messages listed in Table 2. The `got_piece()` method sends HAVE messages to all connected peers on successful download of a piece. While the `SingleDownload` class described previously also performs actions on piece completion, HAVE messages must be broadcast to all connected peers and `SingleDownload` objects only have access to a connection to a single peer.

### 2.5 Related Work

Network connectivity proxying (NCP) have been proposed as a means of letting edge devices sleep and maintain network connectivity [18], [22] and are already available to the public [8]. These network proxies maintain a full network present for the host while it sleeps. In [27], [25], and [26] a power management proxy for Gnutella is explored. By supporting a subset of the Gnutella protocol messages the proxy allows a host peer to sleep while not active but respond to crucial messages and wake up when necessary. The Gnutella proxy was evaluated by measuring file download time and query forwarding rates. File download time was shown to increase from 1 second with a peer that was awake 100% of the time to 9 seconds when the proxy was used, however a significant amount of this increase was due to the transition from sleep to wake in Windows XP. The results indicate that 25% of P2P hosts moving to the proxy would result in savings of $38 million per year in the US. While these savings are based off of the 60 million PCs in US homes, Green BitTorrent targets the much larger market of digital content delivery to homes with digital cable service.

Nano Data Centers [40] have been proposed as a method of reducing the energy consumption of data centers. These nano data centers are in reality P2P application servers running on top of existing ISP controlled gateways in consumers homes. Using traces from Netflix, Youtube, and IPTV a simulation model is built which demonstrates 20% to 30% savings in energy usage when compared to a traditional data center model.

The overall energy efficiency of BitTorrent compared to traditional client server distribution models was explored in [33]. Going under the assumption that peers involved in content distribution will be on anyways, the authors conclude that P2P content distribution is more energy efficient than data centers. With respect to BitTorrent specifically, several proxying solutions have been proposed [1] [5]. The work in this thesis differs dramatically from this approach. The Somniloquy project from Microsoft [1] developed a low power proxying device that handles a subset of application requirements while the main computer sleeps. A BitTorrent client for Somniloquy was created as a proof of concept. Important to note is that this BitTorrent client could
only download and not upload, thus making it of limited use. The low power proxy in [1] specifically handles only download requests which in turn degrades the strength of the swarm by consuming resources without contributing. Similarly, the proposed proxying method in [5] designates a specific machine in a LAN as a proxy which downloads on the behalf of other peers which can then be powered down. The proxy was evaluated via a test bed of several PCs connected to the Internet over a 100 Mbit/s connection downloading files that ranged in size from 3.95 GB to 4.71 GB. In the standard BitTorrent experiments all of the PCs operated as normal while in the proxy experiments one PC served as a proxy for the others. The BitTorrent proxy has two major advantages: a reduction in energy consumed and possible improvement to overall download times. Energy consumption is reduced because only the proxy needs to remain powered on at all times and results indicate savings of up to 95%. Overall download times are reduced due to a reduction in the overall number of peers in a swarm. As the proxy handles downloads for each of the peers behind it the amount of data transferred (via the BitTorrent swarm) is decreased, which increases the per-peer resources available, in turn decreasing download times by approximately 22%. This proxying method is limited by geographic and network topology however. Clearly the peer-proxy relation is dependent on the peers and proxy being on the same LAN. Further, if the proxy is a dedicated device, i.e., not used and thus fully powered on at all times for additional services, the energy savings scale with the number of peers it proxies for. Green BitTorrent is unique in that it is independent of the geographic topology of peers in addition to maintaining the viability of the swarm in which peers participate.

2.6 Chapter Summary

This chapter described methods of content distribution necessary for understanding the remainder of this thesis. Client-server content distribution was described with an emphasis on the energy consumption of data centers. Additionally, peer-to-peer content distribution was examined along with potential energy savings it might provide.

The BitTorrent protocol was introduced and described. Along with the BitTorrent protocol discussion the ns-2 network simulator was described, within which the new Green BitTorrent was implemented. The BitTornado BitTorrent client was also introduced and described. As the BitTornado client is a complex piece of software, class diagrams and explanations of relevant methods and interworking of the components necessary to understand the modifications were provided.
Chapter 3:

Green BitTorrent Protocol Extension

3.1 What is a Green BitTorrent?

The primary concern of a Green BitTorrent is maintaining the fewest powered-on peers at any given time by sleeping when resources are unneeded. Related is the transition to a sleep state not disrupting the peer list of its local swarm, and the ability to be awoken when its resources become needed again. An awake peer should always have a sufficient number of other peers that are awake to download from, thus an awake peer must be able to wake up other sleeping peers.

To achieve the above, new peer states, timers, and events are defined. Standard BitTorrent does not specifically define peer states, however peers can be logically considered to be in one of two states: not connected or connected. The three new peer states that Green BitTorrent defines, are unknown, connected, and sleeping and are described in detail in the next section. In addition to the new peer states events detecting the disconnection of a peer, finding new peers, going to sleep, and waking up will be discussed. Finally, the effects that Green BitTorrent clients have on a swarm of peers that include standard BitTorrent clients will be discussed.

3.2 New Peer States

When a peer first receives a list of other peers from the tracker no statement as to the state of these peers can be made. A peer might be sleeping or it might be awake. Thus, upon gaining new knowledge of the existence of a peer the peer is marked as being in the unknown state. That is, neither sleeping nor awake.

Once a peer begins communicating with another peer the other peer is marked as connected. The connected state implies that a peer has an active TCP connection with the peer that has marked it connected and file pieces can be uploaded and downloaded on the connection. Note that the ability to upload and download is determined by the BitTorrent protocol and not necessarily data availability or the existence of a connection between peers. No transfers will take place unless the connection between the two peers is UNCHOKED and INTERESTED.
Finally, a peer is marked as *sleeping* if it has disconnected with this peer, and thus has closed the TCP connection to it. For further data transfer between the two peers a new TCP connection must be established.

### 3.3 New Peer Events

The detection of disconnected peers already exists in standard BitTorrent. Standard BitTorrent clients eject peers from the peer list that have disconnected and the knowledge of the peer’s existence is lost. This can be seen in Figure 12 line 3, a code snippet from the BitTornado BitTorrent client. In contrast, a disconnected peer is not removed from a Green BitTorrent clients peer list but rather marked as sleeping, as seen in Figure 11.

Green BitTorrent peers must be able to discover and connect to new peers as the make up of their local swarm changes. For example, if all the members of a peer’s local swarm disconnect the peer must have the ability to find and initiate new connections. Standard BitTorrent peers accomplish this by requesting a new set of peers from the tracker at regular intervals and initiating connections with new or already known but not connected peers as necessary. A Green BitTorrent peer must operate somewhat differently on the new peers received from the tracker.

Figure 13 shows the behavior of a Green BitTorrent peer when new connections are needed. After receiving a new set of peers from the tracker the state for each peer in this list is set to *unknown* (line 4). If the number of currently open connections is less than *max_connect* a peer is randomly selected from the peer list. Before connecting to a peer the peer is tested for wake up.
1. on (timeout of connection timer)
2. check with tracker for new peers as needed
3. for (all new peers in peer list)
4. p.state = unknown
5. while (count of connected peers < max connect)
6. p = randomly selected peer in my peer list
8. if (have tested all peers) exit this loop
9. if (wake-up condition == true)
10. send wake-up message to peer p
11. try to connect to peer p
12. if (TCP connection established)
13. p.state = connected
14. else
15. remove peer p from my peer list
16. restart connection timer

Figure 13. Description of changed peer discovery event

1. on (timeout of inactivity timer)
2. send not interested message to connected peers
3. send choke message to connected peers
4. close all of my TCP connections
5. my.state = sleeping
6. enter sleep state

Figure 14. Description of new peer inactivity and sleep event

The specific condition tested is:

(p.state == unknown)
if this peer is a seed, and

((p.state == unknown) OR (p.state == sleeping))
if this peer is a leech.

If the condition is met, a wake up message is sent and an attempt at establishing a TCP connection occurs. This wake up message could be a standard Magic Packet [4] or another packet type. If the TCP connection fails, the peer is considered “dead” (for example, it has been physically removed from the network) and removed from the peer list completely.

Figure 14 shows how a Green BitTorrent peer transitions to sleep. When inactivity is detected a NOT_IN-TERESTED message is sent to all connected peers followed by a CHOKE message (lines 2 and 3). These two messages ensure that the swarm is in a consistent state when a peer goes to sleep. Once these messages have been sent the peer closes all open connections and goes to sleep.
1. on (detection of my wake-up triggered by peer p) 
2. if (TCP connection is established from peer p) 
3. my.state = connected 
4. send my file contents bitfield to peer p 
5. run choking algorithm

Figure 15. Description of new peer wake up event

Figure 15 shows what occurs when a sleeping peer receives a wake up message. Once the inbound TCP connection is established (line 2) the now awake peer sends a BITFIELD message (described in section 2.2 and Table 2) to the peer that sent the wake up message. The sending of the bitfield message enables the peer that sent the wake up message to determine whether or not it is INTERESTED in the pieces the now awake peer has available.

3.4 Backwards Compatibility

While it is expected that Green BitTorrent clients will be used primarily in swarms composed exclusively of other Green BitTorrent clients they are backwards compatible with standard BitTorrent clients. However, without additional measures, Green BitTorrent clients will degrade the performance of standard BitTorrent clients. As a standard BitTorrent client will eject any peer that goes to sleep from its peer list, sleeping Green BitTorrent peers will remain asleep and unusable unless woken up by other Green BitTorrent peers. One way of mitigating this issue involves Green BitTorrent clients detecting the presence of standard BitTorrent clients in a swarm and reverting to standard behavior (i.e., not going to sleep). The methods for client type detection (standard or green), signaling, and any negotiation between clients are beyond the scope of this thesis.

3.5 Chapter Summary

In this chapter a description of a Green BitTorrent is presented. The new peer states required for a Green BitTorrent were described. The three new peer states (unknown, sleeping, connected) allow long lived knowledge of sleeping peers to be retained. The new events in a Green BitTorrent client were also described. The maintenance of peer knowledge after a peer goes to sleep was demonstrated. The mechanism by which a Green BitTorrent peer goes to sleep was explained, specifically the messages that are sent by a peer transitioning to sleep to ensure the swarm remains in a consistent state. What occurs when a peer wakes up was also described. Finally issues of compatibility between Green BitTorrent clients and standard BitTorrent clients were explored.
Chapter 4:
Implementation of Green BitTorrent

The Green BitTorrent client was implemented within the ns-2 network simulator as well as a modification to the BitTornado BitTorrent client. The ns-2 implementation allows for experiments involving a large number of peers that would otherwise not be accessible. The BitTornado implementation allows testing of the Green BitTorrent with real peers actively transferring data. This chapter explains the changes and additions to ns-2 and BitTornado while implementing the Green BitTorrent client.

4.1 Changes to ns-2 BitTorrent Simulation

As the BitTorrent model by Eger et al. was used as a starting point, the code that the Green BitTorrent would necessarily change was first isolated. Due to the nature of the model, the only class that required modification was `BitTorrentApp`.

4.1.1 Peer States

The peer states noted in Section 3.2 were handled by including a new possible value for entries in a given peer’s peer list. While the original model has three possible values, -1 for not connected, 0 for a half completed connection (i.e., one peer asking for a connection but not yet responded to by the other peer), and 1 for fully connected, an additional value 2 indicating a given peer is sleeping was added. The original values of -1 and 1 map directly to the unknown and connected peer states. In the simulation environment the entries in an individual peer’s peer list are shared. I.e., there is only ever one instance of an entry for a given peer and this instance exists in multiple lists.

4.1.2 Peer Events

The event described in Figure 11 was implemented by overriding the base `BitTorrentApp`’s `close_connection()` method. In the original model, `close_connection()` ejected the peer the closed the connection from the peer list while the overridden method marks it as sleeping (a connected value of 2).
The peer discovery event described in Figure 13 was implemented by overriding the `check_connections()` method. The original `check_connections()` attempts to connect to any non-connected peers (connected value = 1) as long as the total number of peers in the local swarm is less than `max_connect`. The new `check_connections()` in `GreenBitTorrentApp` differs in two distinct ways. First, it sends a wakeup message to all peers it attempts to connect to as there is no way to distinguish whether or not a peer in the `unknown` state is sleeping. Second, if a peer’s state is known to be `sleeping` then it will not be sent a wake up message or connected to if this peer is a seed as per the condition specified in Figure 13.

The event described in Figure 14 is implemented via two new methods in `GreenBitTorrentApp` called `try_sleep()` and `go_to_sleep()`. `try_sleep()` is called by a timer which fires every `n` seconds (with `n` being configurable). `try_sleep()` first asserts that the peer trying to sleep is a seed, if it is not no action occurs. If it is a seed, however, the activity of the peer is checked. This entails checking the active requests of each connected peer. If no peer has an active request for data, then the seed is determined to be inactive and a call to `go_to_sleep()` is made. `go_to_sleep()` again asserts that the peer trying to sleep is a seed. If the peer trying to sleep is a seed, then each peer it is connected to (peer state `connected` or connected value = 1) is sent a `NOT интересован` message, choked, and has its TCP/IP connection to the peer going to sleep closed. Additionally, the peer that went to sleep remains unavailable for wake up for a configurable time, simulating the time necessary for a computer to transition to a sleeping state.

The event described in Figure 15 is implemented via two new methods in `GreenBitTorrentApp` called `wake_up` and `done_waking_up()`. As new new connections are always initiated with a wake up call (described in the implementation of Figure 13) `wake_up()` first asserts that the peer is asleep. If the peer is not sleeping, no additional action is taken and the connection proceeds as normal. If it is, then a timer is started (to model the amount of time a computer would require to transition from sleep to awake). When this timer fires, `done_waking_up()` is called. `done_waking_up()` first begins listening on the simulated TCP connection to the peer that woke it up. Once the connection is established the shared peer list entry representing the now awake peer has its connected value set to 1, effectively setting the peers state to connected as per Section 3.2. Next, the newly awake peer sends its bitfield to the peer that woke it up and runs the choking algorithm.
4.2 Changes to BitTornado

The BitTornado code base is more decoupled and modular than the ns-2 model by Eger et al. and thus changes in multiple locations were necessary. Several classes were identified for modification:

- **Torrent** - To be extended via a derived class *GreenTorrent*
- **Encoder** - To be extended via a derived class *GreenEncoder*
- **Connector** - To be extended via a derived class *GreenConnector*

Figure 16 is a class hierarchy diagram illustrating the relationship between parent and child classes and overridden methods.

4.2.1 Peer States

The peer states identified in Section 3.2 were not directly translated to the Green BitTorrent client. The connected state can be directly inferred by the existence of a connection to another peer. The unknown state can be inferred by the absence of a connection to a peer as well as the peer not existing in a sleeping peers list. Finally, the sleeping state is inferred by a peer's membership in a sleeping peers list.
4.2.2 Peer Events

The event described in Figure 11 was implemented in `GreenConnector#connection_lost()`. Unlike the simulation, the state of peers in the peer list are not shared, and so a slightly different technique was used. As seen in Figure 17, `GreenConnector#connection_lost()` adds the disconnected peer to the `sleeping_peers` list and then defers to `Connector#connection_lost()` seen in Figure 12.

The event described in Figure 13 is implemented via methods in the `GreenEncoder` and `GreenConnector` classes. The `Encoder` class is responsible for managing connections to peers, and `GreenEncoder#start_connection()` implements lines 9 through 11 in Figure 13 as seen in Figure 18. Again, as state is not shared between peers in a real swarm, `GreenConnector#connection_made()` handles the setting of the peer states in Section 3.2 as seen in Figure 19 by implementing the functionality in line 13 of Figure 13.

The event described in Figure 14 is implemented in `GreenTorrent#try_to_sleep()` and `GreenTorrent#go_to_sleep()` seen in Figure 20. `GreenTorrent#try_to_sleep()` implements the inactivity timer on line 1 of Figure 14 while `GreenTorrent#go_to_sleep()` implements lines 2 through 6 of Figure 14.

Line 2 of Figure 20 first asserts that this peer is a seed. If not, check for inactivity is rescheduled to occur in another 15 seconds. If this peer is a seed, lines 5 through 8 check for activity by looking for active requests by any connected peers or interest by any connected peers. If an active requests or interested peer exists, the inactivity check is rescheduled to occur in another 15 seconds. Finally, if this peer is a seed and is inactive, line 9 makes a call to `GreenTorrent#go_to_sleep()` on line 9.
def connection_made(self, connection):
    if connection.get_ip() in self.sleeping_peers:
        self.sleeping_peers.remove(connection.get_ip())
    return Connector.connection_made(self, connection)

Figure 19. GreenConnector#connection_made() method

def try_to_sleep(self):
    if not self.is_seed:
        self.rawserver.add_task(self.try_to_sleep, 15)
    return False

for c in self.connector.connections.itervalues():
    if len(c.upload.buffer) > 0 or c.upload.is_interested():
        self.rawserver.add_task(self.try_to_sleep, 15)
    return False

return self.go_to_sleep()

def go_to_sleep(self):
    if self.is_asleep:
        return False
    self.rerequest.stopped = True
    self.encoder.close_all()
    self.rawserver.sockethandler.shutdown()
    self.is_asleep = True
    return True

Figure 20. GreenTorrent#try_to_sleep() and GreenTorrent#go_to_sleep() methods

def wake_up(self):
    if not self.is_asleep:
        return False
    self.is_asleep = False
    self.rerequest.stopped = False
    self.rawserver.add_task(self.try_to_sleep, 15)
    self.rerequest.start()
    return True

Figure 21. GreenTorrent#wake_up() method

GreenTorrent#go_to_sleep() first asserts that this peer is a seed. Assuming this peer is a seed, lines 14 through 17 of Figure 20 implement lines 2 through 5 of Figure 14 and the peer transitions to a sleep state. The event described in Figure 15 is implemented in the GreenTorrent#wake_up() method seen in Figure 21.

4.3 Chapter Summary

This chapter described the implementation of the Green BitTorrent client within both the ns-2 network simulator and the BitTornado BitTorrent client. The points of application control that must be modified to
allow the decoupling of TCP/IP connection state from application state as described in Section 3.1 were isolated. The implementation of the changes and how they relate to the peer states and events described in Sections 3.2 and 3.3 were also described. Finally, a class hierarchy to illustrate the relationship between the standard BitTorrent client and the Green BitTorrent modifications was provided.
Chapter 5:
Evaluation of Green BitTorrent

In order to test the efficacy of Green BitTorrent swarms under various conditions must be evaluated. Recognizing the scenarios under which energy savings are possible and how much savings are achievable is the primary concern of the evaluation of Green BitTorrent. To this end a typical large Green BitTorrent swarm in the simulator was evaluated and compared the results to a standard BitTorrent swarm under the same conditions. A smaller swarm of clients using the real Green BitTorrent client was then tested and compared to the results of a standard BitTorrent swarm. Finally, the simulation was validated by running experiments with the same swarm makeup under which the real clients were evaluated and compared the results.

5.1 Simulation Experiments

The work in this section originally appeared in [14]. In section 5.2.2.1 new results from the real client as well as a validation of the simulation experiments are provided. A BitTorrent system can be viewed as having control and response variables to be manipulated and measured, respectively.

The control variables for the system configuration were:

- Number of swarms a peer participates in
- Number of peers to maintain in a swarm
- Number of connected peers (max_connect)
- Number of seed peers at the start of the swarm
- Number of leech peers
- File size
- Upload and download bandwidth for peers
- RTT between peers
The control variables for a Green BitTorrent peer were:

- Transition time to wake-up and go to sleep
- Connection timer preset value
- Inactivity timer preset value

The control variables for system workload were:

- Interarrival time distribution for peers entering a swarm
  - Distribution parameters including mean interarrival time \(T_{arrival}\)

The response variables for a peer were:

- Sleep time
- Awake time
- File download time

### 5.1.1 Description of Experiment

An experiment to measure file download, sleep, and awake times as a function of the interarrival time of peers into a swarm was designed. The system modeled a single swarm of 50 peers (a typical large swarm) with one additional initial seed peer (this initial seed peer was set to never sleep in any of the cases) and the rest of the entering peers initially containing no pieces. A max\_connect value of 5 (the typical value for a BitTorrent client) was used. A file size of 1GB corresponding to a small video file was used. File pieces were 256 KB. An upload data rate of 2 Mb/s and download data rate of 10 Mb/s per peer corresponding to Verizon residential FIOS 2008 base rates [41] was used. The RTT between peers was 10 ms modeling a swarm contained within the domain of a single ISP. The experiments were conducted on the high performance computing resources of the University of South Florida’s Research Computing.

Parameters specific to Green BitTorrent were set as follows. The wake-up and go to sleep transition times were 300 ms each. This is a reasonable time for an operating system to save its state and for a processor to recover this state and resume execution. The Linux-based OLPC machine requires only tens of milliseconds to wake-up [42]. The connection timer was set to 300 s (5 min). This is the default connection timer value used in BitTorrent client implementations for checking with the tracker for new peers. The inactivity timer was set to 15 s. This value was selected as reasonable to prevent oscillation between awake and sleep.
Peers arrived into a swarm as a Poisson process (that is, interarrival times are exponentially distributed). This models independent human behavior well. The mean interarrival times were varied from very short to model a flash crowd to very long to model peers arriving into a swarm where all other peers had already completed their downloads (that is, all “old” peers were seeds). The mean interarrival times used were, $T_{\text{arrival}} = 0, 0.5, 1, 2, 4, 8, 16, \text{ and } 32 \text{ minutes}$. The time to download a 1 GB file at 10 Mb/s is about 14.3 min, thus the 16 min and 32 min mean interarrival times will have most, if not all, peers arriving into a swarm where all other peers have already completed their download. Two systems (or cases) were modeled - a control network consisting of standard BitTorrent where no peers could sleep and a green BitTorrent network. For each interarrival time 30 replications were run, each replication with a different initial random number seed. The average of the replications was computed and plotted.

### 5.1.2 Results from Experiment

Three graphs for standard BitTorrent (Figures 22, 23, and 24) and three for Green BitTorrent (Figures 25, 26, and 27) show the results from the experiment. The download times for the 1st, 25th, and 50th peer to enter the swarm as a function of interarrival time are shown in Figures 22 and 25. Awake and sleep time for all peers with a mean interarrival time of 16 min are seen in Figures 23 and 26. Figures 24 and 27 show the sleep and awake time for peer 25 over all interarrival times with the gray showing the sleep time. The difference in download times between the standard and Green BitTorrent simulations for peers 1, 25, and 50 are shown in Table 3.

From Figures 22 and 25 it can be seen that:

- The mean download time for client #1 is high and fairly constant independent of the client interarrival time and case.

- The mean download time for clients #25 and #50 decreases as the interarrival time increases for the standard BitTorrent and Green BitTorrent.

- Green BitTorrent has larger file download times than standard BitTorrent. Table 3 shows the percentage increase in download time for green BitTorrent compared to standard BitTorrent.
From Figures 23 and 26 it can be seen that:

- There is no sleep time for standard BitTorrent, but there is considerable sleep time for Green BitTorrent.
  For standard BitTorrent the sum of the awake times for the entire swarm is 324.6 hours, for green BitTorrent it is 72.1 hours (253.8 hours is now spent in sleep). This represents an energy savings of 77.8% measured over all peers.

From Figures 24 and 27 it can be seen that:

- There is no sleep time for standard BitTorrent, but there is considerable sleep time for Green BitTorrent.
  For standard BitTorrent the sum of the awake times for client #25 for all interarrival times is 29.6 hours, for Green BitTorrent it is 10.2 hours (19.9 hours is now spent in sleep). This represents an energy savings of 65.5% measured over all interarrival times.
Figure 24. Sleep time for peer 25 standard BitTorrent (simulation)

Figure 25. Download time for Green BitTorrent (simulation)

Figure 26. Sleep time for Green BitTorrent (simulation)
Table 3. Increase in download time for Green BitTorrent versus standard BitTorrent (simulation)

<table>
<thead>
<tr>
<th>Client number</th>
<th>0 min</th>
<th>0.5 min</th>
<th>1 min</th>
<th>2 min</th>
<th>4 min</th>
<th>8 min</th>
<th>16 min</th>
<th>32 min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client #1</td>
<td>4.8%</td>
<td>1.5%</td>
<td>1.8%</td>
<td>1.9%</td>
<td>1.9%</td>
<td>1.9%</td>
<td>2.1%</td>
<td>1.6%</td>
</tr>
<tr>
<td>Client #25</td>
<td>6.7%</td>
<td>3.7%</td>
<td>-0.7%</td>
<td>5.2%</td>
<td>11.0%</td>
<td>44.8%</td>
<td>30.9%</td>
<td>18.1%</td>
</tr>
<tr>
<td>Client #50</td>
<td>7.0%</td>
<td>4.6%</td>
<td>6.0%</td>
<td>15.5%</td>
<td>32.8%</td>
<td>18.7%</td>
<td>9.4%</td>
<td>8.2%</td>
</tr>
<tr>
<td>Average for all</td>
<td>4.9%</td>
<td>3.4%</td>
<td>3.5%</td>
<td>5.2%</td>
<td>13.7%</td>
<td>23.0%</td>
<td>22.0%</td>
<td>18.3%</td>
</tr>
</tbody>
</table>

5.1.3 Discussion of Results

Two key questions emerge from the observed results, they are:

1. Why does client #1 always have a large download time that is roughly the same for Green and standard BitTorrent?

2. What explains the slightly larger download time for Green BitTorrent as compared to standard BitTorrent?

For the first question, client #1 always arrives into a swarm with only one initial seed present. Later clients arrive into a swarm with many peers present (of which many are also seeds by virtue of already having completed their download). When the client interarrival time is small, other peers will arrive and be present during the download period for client #1. Thus, client #1 can download from multiple peers in the case of small interarrival times, but not so when interarrival times are large. A slight reduction in download time can be seen for the small interarrival times (but, at these times there will also be competition from other peers preventing client #1 from getting a full download rate from the other peers).
For the second question, the larger download times for Green BitTorrent can be explained as seeds going to sleep putting Green BitTorrent at a disadvantage with regards to download time (compared to standard BitTorrent). As there are fewer awake peers in the Green BitTorrent swarm, there are also fewer peers that might initiate a new inbound connection. While the number of outbound connections is limited, inbound connections (and thus the total number of connections) are effectively not limited. This reduces the overall possible bandwidth available to a given peer, however it is extremely dependent on the make-up of the swarm at any given time. The time to wake-up sleeping seeds also factors into the increased download time for Green BitTorrent.

5.2 Real Client Experiments

In order to validate the simulations experiments using the real client were run. The real client experiments utilized a swarm of 10 peers, with one additional peer acting as the initial seed. The clients were each hosted on separate computers. Each computer was configured with an Intel® Xeon® X3220 quad-core CPU running at 2.40 Ghz, 4 GB of ram, and an Intel® PRO/1000 network adapter. A photograph of the computers can be seen in Figure 28. While these peers were connected to each other over a 1000 Mbit switch they were bandwidth limited to 10 Mbits/second download and 2 Mbits/second upload via the built in bandwidth limiter of the BitTornado client and distributed a 100 MB file.

In addition to implementing Green BitTorrent, an RPC server that allows the experiments to be controlled was also created. The RPC server uses the standard Python xmlrpclib, a library that implements the XML-RPC remote procedure call mechanism, and exposes a set of commands that can be issued to clients:

- `create_torrent()`
- `start_torrenting()`
- `stop_torrenting()`
- `get_stats()`
- `get_metainfo()`
- `go_to_sleep()`
- `wake_up()`
create_torrent() creates initializes the client by creating a Torrent object of the specified types. For experimentation purposes the creation of “Torrent” and “GreenTorrent” torrents is allowed with “Torrent” being a standard BitTorrent and “GreenTorrent” being a GreenBitTorrent. Creating a torrent involves reading the torrent file and initializing data blocks on the machine the client is running.

After issuing a create_torrent() command, a call to start_torrenting() is made. start_torrenting() causes the client to connect to the tracker and become a member of the swarm. With no parameters start_torrenting() will immediately connect to the tracker and join the swarm. There is an optional parameter that causes a delay (of user supplied seconds) before contacting the tracker and joining the swarm. To shut down a client a call to stop_torrenting() is made. stop_torrenting() runs through the normal client shut down procedure and returns statistics for the client’s session.

The statistics for the clients session are exposed via a call to get_stats(). The specific statistics returned are governed by the underlying Torrent object. For standard Torrent objects the statistics returned are:

- seed - whether or not this Torrent is a seed
- upTotal - the total amount of data uploaded by this Torrent
- downloadTime - the time (in seconds) this Torrent has spend downloading data
- totalTime - the total time (in seconds) this Torrent has been a member of the swarm
• **percentDone** - the percentage of the total file this *Torrent* has downloaded

In addition to the above, *GreenTorrent* objects also return the following:

• **sleepTime** - the time (in seconds) this *GreenTorrent* has spent sleeping

• **awakeTime** - the time (in seconds) this *GreenTorrent* has spent awake

Related to *get_stats()* and *get_metainfo()* returns the human readable portion of the metainfo (torrent file) that a client is working on.

Finally, *go_to_sleep()* forces a *GreenTorrent* client to go to sleep and *wake_up()* wakes a *GreenTorrent* client up.

### 5.2.1 Description of Experiment

As in the simulation experiments, wake-up and sleep transitions were set to 300 ms and an inactivity time of 15s triggered a transition from awake to sleep. Again file download, sleep, and awake times as a function of interarrival time were measured. All other values were left at the defaults. An ideal transfer time (if the download bandwidth of a peer is completely saturated) for these parameters is 80 seconds, and thus interarrival times to be a range on either side of this value were chosen. The interarrival times chosen were \( T_{\text{arrival}} = 0, 40, 120, 150, \) and 400 seconds.

### 5.2.2 Results from Experiment

Three graphs for standard BitTorrent (Figures 29, 30, and 31) and three for Green BitTorrent (Figures 32, 33, and 34) show the results from the experiment. The download times for the 1st, 5th, and 10th peer to enter the swarm as a function of interarrival time are shown in Figures 22 and 32. Awake and sleep time for all peers with a mean interarrival time of 16 min are seen in Figures 30 and 33. Figures 31 and 34 show the sleep and awake time for peer 55 over all interarrival times with the gray showing the sleep time. The difference in download times between the standard and Green BitTorrent simulations for peers 1, 5, and 10 are shown in Table 4.
From Figures 29 and 32 it can be seen that:

- The mean download time for client #1 is high and fairly constant independent of the client interarrival time and case.
- The mean download time for clients #5 and #10 decreases as the interarrival time increases for the standard BitTorrent and Green BitTorrent.
- Green BitTorrent has larger file download times than standard BitTorrent. Table 4 shows the percentage increase in download time for Green BitTorrent versus standard BitTorrent.

From Figures 30 and 33 it can be seen that:

- There is no sleep time for standard BitTorrent, but there is considerable sleep time for Green BitTorrent. For standard BitTorrent the sum of the awake times for the entire swarm is 5.71 hours, for green BitTorrent it is 1.31 hours (3.85 hours is now spent in sleep). This represents an energy savings of 74.6% measured over all peers.

From Figures 31 and 34 it can be seen that:

- There is no sleep time for standard BitTorrent, but there is considerable sleep time for Green BitTorrent. For standard BitTorrent the sum of the awake times for client #5 for all interarrival times is 1.30 hours, for Green BitTorrent it is 0.61 hours (0.63 hours is now spent in sleep). This represents an energy savings of 50.8% measured over all interarrival times.
Figure 30. Sleep time for standard BitTorrent

Figure 31. Sleep time for peer 5 standard BitTorrent

Figure 32. Download time for Green BitTorrent
Total awake = 1.31 hrs
Total sleep  = 3.85 hrs

Figure 33. Sleep time for Green BitTorrent

Total awake = 0.61 hrs
Total sleep  = 0.63 hrs

Figure 34. Sleep time for peer 5 Green BitTorrent

Table 4. Increase in download time for Green BitTorrent compared to standard BitTorrent

<table>
<thead>
<tr>
<th>Mean client interarrival time</th>
<th>0 sec</th>
<th>40 sec</th>
<th>120 sec</th>
<th>150 sec</th>
<th>400 sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client number</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client #1</td>
<td>1.4%</td>
<td>-0.4%</td>
<td>-1.3%</td>
<td>-0.8%</td>
<td>0.5%</td>
</tr>
<tr>
<td>Client #5</td>
<td>1.6%</td>
<td>-1.7%</td>
<td>-22.1%</td>
<td>0.4%</td>
<td>24.4%</td>
</tr>
<tr>
<td>Client #10</td>
<td>-0.5%</td>
<td>4.6%</td>
<td>13.7%</td>
<td>-8.7%</td>
<td>3.8%</td>
</tr>
<tr>
<td>Average for all</td>
<td>1.2%</td>
<td>0.6%</td>
<td>-7.1%</td>
<td>-1.8%</td>
<td>1.3%</td>
</tr>
</tbody>
</table>

44
5.2.2.1 Simulation Validation

To validate the simulation, simulated experiments were run with the same parameters as the real experiments. Again, three graphs for standard BitTorrent (Figures 35, 36, and 37) and three for Green BitTorrent (Figures 38, 39, and 40) show the results from the experiment. The download times for the 1st, 5th, and 10th peer to enter the swarm as a function of interarrival time are shown in Figures 22 and 38. Awake and sleep time for all peers with a mean interarrival time of 16 min are seen in Figures 36 and 39. Figures 37 and 40 show the sleep and awake time for peer 5 over all interarrival times with the gray showing the sleep time.
Figure 37. Sleep time for peer 5 standard BitTorrent (simulation validation)

Figure 38. Download time for Green BitTorrent (simulation validation)

Figure 39. Sleep time for Green BitTorrent (simulation validation)
5.2.3 Discussion of Results

The results show that a real Green BitTorrent client is capable of significant energy savings. Of question is whether or not the results of the smaller test bed will scale up. This question is answered by the validation results in Section 5.2.2.1.

While the numbers are not identical between the simulation and real experiments this is to be expected. There are numerous variables that come into play in real world experiments, including, but not limited to the load on the machines, the load on the networking devices connecting the machines, and subtle differences between implementations. Any of these can cause a swarm that might start identically in both the simulation and real experiments to evolve in quite a different fashion. In particular, the bandwidth limiter in the real client is not perfect. It uses a rolling average and delays sending and requesting pieces as opposed to actually limiting the rate at which data is sent. Additionally, the simulation was noted to behave in unexpected ways when peers were given larger amounts of bandwidth, indicative of a systematic error. The difference in download times between real and simulation clients can be seen in Table 5. Of note is that the real client outperformed the simulation in each case. What is important, however, is a wide view of the results. The shape of the graphs in Figures 29 through 40 are all similar and thus while the absolute numbers
The savings seen in Figures 37 and 40 is 45.2%, which is very close to the 50.8% savings seen in Figures 31 and 34. Finally, the 70.1% savings seen for all peers in the 400 second interarrival time seen in Figures 36 and 39 are again well within 10% of the 74.6% savings seen in Figures 31 and 34.

5.3 Chapter Summary

This chapter explained the experimental setup describing both simulation and real experiments. Results for two sets of experiments were discussed:

1. A simulation experiment of a swarm of 50 peers (plus 1 seed) transferring a 1 GB file.

2. A real client experiment of a swarm of 10 peers (plus 1 seed) transferring a 100 MB file.

The results of the experiments indicated significant savings (up to 77.8%) in energy with minimal download penalty (a worst case of 23.0% increase in download times on average).
Chapter 6:

Summary and Future Work

It has been shown that relatively simple changes to BitTorrent can achieve a more energy efficient operation - a Green BitTorrent. These changes allow peers to sleep without being dropped from peer lists. Effectively, TCP connection state has been decoupled from peer state.

6.1 Green BitTorrent Energy Savings

Green BitTorrent was shown to consume less than 25% of the energy as standard BitTorrent (where all clients are fully powered-on 24/7) with only modest penalty in increased download time. For small interarrival times, file download time is increased by less than 10% (as seen in Table 4). For medium and large interarrival times, download time for an individual peer is increased by up to about 25%, but typically by much less (as seen in Table 4). Of note however is that the difference in download time for all peers was between -7.0% and 1.3% for each interarrival time. The overall energy savings achievable with Green BitTorrent if clients sleep 75% of the time could be over $1.6 billion per year in the US alone if 100 million file sharing units, each consuming a 25 W on average (and assuming a residential electricity rate of $0.10 per kWh) is assumed. This level of savings appears to be feasible based on the methods developed and evaluated in this paper.

6.2 Directions for Future Research

While considerable energy savings have been shown, the implementation is currently “dumb”. Peers are only aware of the states of other peers as described in Section 3.2. This ignores an entire taxonomy related to energy efficiency. For example, the energy costs in various locales are not equal, nor are they produced in the same manner [37]. A “smarter” Green BitTorrent client could favor waking up peers that are using “inexpensive” energy via changes to the peer discovery algorithm described in Figure 13. Similarly a smarter Green BitTorrent client could favor peers using “expensive” energy via changes to the choking algorithm allowing them to finish downloads faster and thus spend less time awake. Figure 13 could be altered to wake peers up based on energy costs (both monetary and environmental). The choking algorithm could be modified
to give a higher ordering (and thus more bandwidth and quicker download times) to peers using expensive energy sources, thus including energy in the resource exchange mechanism that makes BitTorrent unique.

Modifications to the tracker could further the goal of energy savings. The tracker is a prime target for examining energy efficiency as peers learn of the existence of other peers through queries to it. The tracker could easily be modified to influence the local swarm of any peer based on energy costs. For example, peers utilizing expensive energy sources could be “funneled” into connecting to peers using cheap energy sources to minimize overall energy consumption.

These future directions can provide a basis for research into future peer-to-peer content distribution systems, in particular the concept of energy as an incentive for resource sharing resulting in highly scalable and energy efficient content distribution.
References


