

A CASE STUDY IN MODELING LARGE-SCALE PEER-TO-PEER FILE-SHARING NETWORKS USING DISCRETE-EVENT SIMULATION

Christopher D. Carothers
Ryan LaFortune

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180, U.S.A.

email: {chrisc, laforr}@cs.rpi.edu

William D. Smith
Mark Gilder

Global Research Center
General Electric

Niskayuna, NY 12309, U.S.A.

email: {smithwd, gilder}@crd.ge.com

ABSTRACT

Peer-to-Peer file-sharing networks are garnering a significant share of home broadband networks. Since 2001, BitTorrent has been one of the most popular protocols for file-sharing. What makes BitTorrent so unique compared to past efforts is that it provides a builtin mechanism to ensure the fair distribution of content and prevents selfishness on the part of peers using game theoretical “tit-for-tat” piece distribution algorithms. While this is good for home consumers, Internet service providers are not pleased with how BitTorrent has overloaded their networks. To better understand the BitTorrent protocol, we have created a detailed simulation model that is able to scale to 10’s of thousands of peers using commodity hardware. In this paper, we describe our model design and implementation approach as well as present a validation and performance study of the model. A key result from this study is that our model’s memory footprint ranges between 67 KB to 2.3 MB per client peer depending on the simulation configuration parameters. This is 30 to 1000 times less memory than required by the operational BitTorrent software.

1 INTRODUCTION

Peer-to-Peer (P2P) file-sharing is the “killer application” for the consumer broadband Internet. CacheLogic’s (CacheLogic 2005) monitoring of tier 1 and 2 Internet service providers (ISPs) in June, 2004 reports that between 50% and 80% of all traffic is attributed to P2P file-sharing. In 2005 those numbers appear to have been holding steady at 60% of all network traffic on the reporting consumer-oriented ISPs (CacheLogic 2005). The total population logged onto the major P2P networks at any instance in time is about 8 million users who

are sharing some 10 million GB (e.g., 10 Petabytes) of data. This accounts for nearly 10% of the broadband connections world-wide and this trend is expected to grow (CacheLogic 2004).

Much of the data being exchanged on P2P networks is greater than 100 MBs and is largely video files. For example, a typical DIVX format movie is 700 MB, a complete DVD movie can be 4 GB or higher (single layer) and the latest high definition movies may require 10 GB or more. With high definition movies (i.e., HD-DVD and Blu-Ray formats) just entering the home theater market, one can expect downloadable content sizes to grow by a factor of 100 to even 1,000, thus pushing the networks traffic loads to much higher levels.

So then, one might ask, *what is the driving force behind these trends?* In addition to the attraction to “free” and/or “pirated” content, a key driving force is the *content distribution economics*. From both a content publisher’s as well as content consumer’s point-of-view, P2P makes good economic sense especially in the context of what has been called the “flash crowd” effect. Here, a single piece of data, such as a new online movie release is so popular, that the number of people attempting to download that file will overload the capacity of the most powerful single site web server. However, in a current generation P2P network, such as BitTorrent, a single low bandwidth host will “seed” content to a massive “swarm” of peers. The hosts within the swarm will then disseminate parts of the content to each other in a peer exchange fashion. This is the heart of how a BitTorrent “swarm” operates. As one peer is obtaining new content, it is simultaneously sharing its content with other peers. Unlike the client-server approach, as the swarm grows, the aggregate network bandwidth of the swarm grows. Thus, from the view point of each node, the data rates are much faster, there is no denial of service on the

part of the content source, and the content source provider's computation and network load remains relatively low.

1.1 Users Happy, ISPs Not

There appears to be a wrinkle in this nirvana of efficient data exchange. Consumer-oriented ISPs are not pleased with how their networks are being used by these peering overlay networks. The cost to them is prohibitive – *on the order of \$1 billion U.S. dollars* (CacheLogic 2005), and the ISPs are not making any additional revenue from these network intensive applications. Consider the on-going case of Shaw Communications Inc, a large ISP for much of western Canada. According to (Grant 2005), Shaw has purchased Ellacoya Networks packet shaping technology to throttle the delivery of P2P network data and ultimately reduce the load on their networks. In effect, current ISP networks were never provisioned for P2P overlay protocols. The amount of bandwidth and routing protocol computations these applications generate make them appear as an “Internet Worm” like Code-Red (Cowie 2002), from the ISP's point of the view. So if you ask, *Is P2P good for the Internet?*, the answer depends greatly on who you ask.

Based on the above motivations, the grand goal of our research project is to better understand the real impact P2P overlay software has on Internet network resources from the distributor, ISP, and end-user point of views. In particular, we focus our research on the *BitTorrent* protocol. The BitTorrent protocol has been one of the most popular P2P file-sharing technologies, with a number of different client implementations (Slyck 2006) and an estimated user-population on the order of 60 million (Shaw 2006) which single-handedly accounted for 50% of all Internet traffic on U.S. cable operator's networks in 2004 (CacheLogic 2004). However, because of content owners shutting down illegal tracker servers due to copyright infringement, usage of BitTorrent has waned to 18% (Grant 2005) and attention has shifted to the eDonkey2K network. In a single 24 hour session, the eDonkey2k overlay network is shown to account for more than 60% of all traffic (including HTTP and mail) through a specific Tier 1 ISP (CacheLogic 2005). The dramatic shift is attributed eDonkey's fully decentralized P2P file-sharing algorithm, which makes it much harder to find and shutdown illegal peer networks. Moreover, content provider / distributors cannot keep unwanted or illegal content out of the peering network. BitTorrent, on the other hand, remains a viable technology for legal content distribution since the tracker is a central point that allows content owners to ensure legitimate content (BitTorrent, 2006). Thus, we believe “BitTorrent-like” applications hold great promise as a legitimate peer-to-peer distribution technology.

1.2 Related Work

The current approaches to studying this specific protocol are either direct measurement of operational BitTorrent “swarms” during a file-sharing session (Legout et al. 2005; Pouwelse 2005) or by real experimentation on physical closed networks, such as PlanetLab (Peterson 2002). The problem with using PlanetLab as a P2P testbed is that the usage polices can limit our ability to explore network behaviors under extreme conditions. That is, your application cannot interfere with other participants in the research network (PlanetLab 2006) and lacks the resources to examine swarm-behaviors at the scale we would like to investigate. These are precisely the extreme cases we wish to examine. Additionally, real P2P Internet measurement studies are either limited in terms of data that they are able to collect because network blocking issues related to network address translations as in the case of (Pouwelse 2005) or limited to active “torrents” in the case of (Legout et al. 2005). Other techniques not necessarily specific to BitTorrent include complex queuing network models, such as (Ge 2003).

While both measurement and queuing network models are highly valuable analytic tools, neither allows precise control over the configuration for the system under test, which is necessary to understand the cause and effect relationships among all aspects of a specific protocol like BitTorrent. For this level of understanding, a detailed simulation model is required. However, the core of any simulation is that - by definition - it is a “falsehood” for which we are trying to extract some “truths”. Thus, the modeler must take extreme care in determining factors which can and cannot be ignored. In the case of BitTorrent, there have been some attempts by Microsoft to model the protocol in detail (Bharambe 2005; Gkantsidis 2005). However, these models have been dismissed by the creator of BitTorrent, Brahm Cohen, as not accurately modeling the true “tit-for-tat”, non-cooperative gaming nature of the protocol as well as other aspects (LeMay 2005).

A third approach is direct emulation of the operational BitTorrent software such as done by (Bindal et al. 2006). Here, the peer code is “fork lifted” from the original implementation. Results are presented using only 700 peers (i.e. cable modems). It is unclear which parts of the BitTorrent implementation were left intact as well as data structure layout changes and so comparisons between this approach and ours in terms of memory and computational efficiency are not possible.

1.3 Contributions

Here, we model all the core decision components of the BitTorrent protocol based on a detailed reading of the source code implementation (BitTorrent Source Code 2006). From this, our contributions are:

1. A memory efficient discrete-event model of the BitTor-

rent protocol built on the ROSS discrete-event simulation system (Carothers et al. 2000; Carothers et al. 2002). The key contribution here is a drastic reduction in model memory usage over the real BitTorrent client. The memory consumption of a single BitTorrent client instance is on the order of 70,000 KB or 70 MB. Our per client model only consumes 67 KB to 2.3 MB per client.

2. A *piece* level data model that insures model protocol accuracy over much higher level flow models yet avoids the event population explosion problem of a typical network packet-level model, such as employed with NS (Network Simulator 2006).
3. Validation study of our BitTorrent model against instrumented BitTorrent operational software as well as previous measurement studies.
4. Model performance results and analysis for a large BitTorrent client scenarios ranging from 128 to 128,000 clients with files up to 4,092 pieces (i.e., 1 GB movie file).

Through this ongoing investigation, we hope to gain insights that will enable better P2P systems that are considered both fair and efficient by not only the users, but the ISPs as well.

2 THE BITTORRENT PROTOCOL

This protocol creates a virtual P2P overlay network of peers using five major components: (i) *torrent files*, (ii) *a website*, (iii) *tracker server*, (iv) *client seeders*, and (v) *client leechers*.

A *torrent* file is composed of a header plus a number of SHA1 block hashes of the original file, where each block or *piece* of the file is a 256KB chunk of the whole file, and further broken down as 16KB sub chunks which will be described further in the section below. The header information denotes the IP address or URL of the tracker for this torrent file. Once created, the torrent file is then stored on a publicly accessible website, from which anyone can download. Next, the original content owner/distributor will start a BitTorrent client that already has a complete copy of the file along with a copy of the torrent file. The torrent file is read and because this BitTorrent client has a complete copy of the file, it registers itself with the *tracker* as a *seeder*. When a *leecher* client finds the torrent file from the publicly available website and downloads it, the BitTorrent client software will also parse the header information as well as the SHA1 hash blocks. However, because it does not have a copy of the file, it registers itself with the *tracker* as a *leecher*. Upon registering, the *tracker* will provide a *leecher* with a randomly generated list of peers. Because of the size of the peer set (typically 50 peers) and the random peer selection, the probability of creating an isolated clique in the

overlay network graph is extremely low, which ensures robust network routes for piece distribution. The downside to this approach is that topological locality is completely ignored which results in much higher network utilization (i.e., more network hops and consumption of more link bandwidth). So, there is a robustness for locality tradeoff being made here.

The *seeder* and other *leechers* will begin to transfer *pieces* (256 KB chunks) of the file among each other using a complex, non-cooperative, *tit-for-tat* algorithm. After a piece is downloaded, the BitTorrent client will validate that piece against the SHA1 hash for that piece. Again, the hash for that piece is contained in the torrent file. When a piece is validated, that client is able to share it with other peers who have not yet obtained it. Pieces within a peer-set are exchanged using a *rarest piece first* policy which is used exclusively after the first few randomly selected pieces have been obtained by a *leecher* (typically four pieces but this is a configuration parameter). Because each peer announces to all peers in its peer-set each new piece it has obtained (via a HAVE message), all peers are able to keep copy counts on each piece and determine within their peer-set which piece or pieces are rarest (i.e., lowest copy count). When a *leecher* has obtained all pieces for the file, it then switches to being a pure *seeder* of the content. At any point during the piece/file exchange process, clients may join or leave the peering network (e.g., *swarm*). Because of the highly volatile nature of these swarms, a peer will re-request an updated list of peers from the *tracker* periodically (typically every 300 seconds). This ensures the survival of the swarm assuming the tracker remains operational.

More recently, BitTorrent has added a distributed hash table (DHT) based tracker mechanism. This approach increases swarm robustness even in the face of tracker failures. However, DHTs are beyond the scope of our current simulation investigation.

2.1 Message Protocol

The BitTorrent message protocol consists of 11 distinct messages as of version 4.4.0 with additional messages being added to the new 4.9 version which is currently in beta testing. All intra-peer messages are sent using TCP whereas peer-tracker messages are sent using HTTP additionally peers upon entering a swarm are in the *choked* and *not interested* states. Once a peer has obtained its initial peer-set (up to 50 peers by default) from the tracker, it will initiate a HANDSAKE message to 40 peers by default. The upper bound on the number of peer connections is 80. Thus, each peer keeps a number of connection slots available for peers who are not in its immediate peer-set. This reduces the probability that a clique will be created. The connections are maintained by periodically sending KEEP ALIVE messages.

Once two-way handshaking between peers is complete, each peer will send the other a BITFIELD message which contains an encoding of the pieces that peer has. If a peer

has no pieces, no BITFIELD message is sent. Upon getting a BITFIELD message, a peer will determine if the remote peer has pieces it needs, if so, it will schedule an INTERESTED message. The remote peer will process the INTERESTED message by invoking its *choker algorithm* which is described next. The output from the remote peer's choker (upload side) is an UNCOKE or CHOKE message. The response to an INTERESTED message is typically nothing or an UNCHOKER message. Once the peer receives an UNCHOKER message, the *piece picker* algorithm (described below) is invoked on the download side of the peer and a REQUEST message will be generated for a piece and 16 KB offset within that piece. The remote will respond with a PIECE message which contains the 16 KB chunk of data. This response will in turn result in additional REQUESTS being sent.

When all 16 KB chunks within a piece have been obtained, the peer will send a HAVE message to all other peers to which it is connected. With receipt of the HAVE message, a remote peer may decide to schedule an INTERESTED message for that peer which results in a CHOKE message and then REQUEST and PIECE messages being exchanged. Thus, the protocol ensures continued downloading of data among all connected peers. Now, should a peer have completely downloaded all content available at a remote peer, it will send a NOT INTERESTED message. The remote peer will then schedule a CHOKE message if the peer was currently in the unchoke state. Likewise, the remote peer will periodically *choke* and *unchoke* peers via the *choker* algorithm. Last, when a peer has made a request for all pieces of content, it will enter *endgame* mode. Here, requests to multiple peers for the same piece can occur. Thus, a peer will send a CANCEL message for that piece to those other peers when one peer has responded with the requested 16 KB chunk.

In order to reduce the complexity of our model, we do not include either KEEP ALIVE or CANCEL messages. In the case of CANCEL messages, they are very few and do not impact the overall swarm dynamics (Legout et al. 2005).

2.2 Choker Algorithms

There are two distinct choker algorithms, each with very different goals. The first is the choker algorithm used by a pure seeder peer. Here, the goal is not to select the peer whose upload data transfer rate is best but instead maximize the distribution of pieces. In the case of leecher peers, it uses a sorted list of peers based on upload rates as the key determining factor. That is, it wants to try to find the set of peers with whom it can best exchange data with. Both choker algorithms are scheduled to run every 10 seconds and can be invoked in response to INTERESTED/NOT INTERESTED messages. Each invocation of the choker algorithm counts as a round. There are three distinct rounds which both choker algorithms cycle through. We begin with the details for the seeder choker algorithm (SCA).

SCA only considers peers that have expressed interest and have been unchoked by this peer. First, the SCA orders peers according to the time they were last unchoked with most recently unchoked peers listed first within a 20 second window. All other peers outside that window are ordered by their upload rate. In both cases, the fastest upload rate is used to break ties between peers. Now, during two of the three rounds, the algorithm leaves the first three peers unchoked as well as unchokes another randomly selected peer. This is known as the optimistic unchoked peer. During the third round, the first four peers are left unchoked and the remaining peers are sent CHOKE messages if they are currently in the unchoke state.

For the leecher choker algorithm (LCA), at the start of round 1, (i.e., every 30 seconds), the algorithm chooses one peer at random that is choked and interested. As in SCA, this is the optimistic unchoked peer (OUP). Next, the LCA orders all peers that are interested and have at least one data block that was sent in the last 30 second time interval, otherwise that peer is considered to be *snubbed*. Snubbed peers are excluded from being unchoked to prevent free riders and ensure that peers share data in a relatively fair way. From that order list, the three fastest peers along with the OUP are unchoked. If the OUP is one of the three fastest, a new OUP is determined and unchoked. Note here, that the OUP could be *not interested*. As soon as it becomes *interested*, this will invoke the choker algorithm as part of INTERESTED message processing.

2.3 Piece Picker

The *piece picker* is a two phase algorithm. The first phase is *random*. When a leecher peer has no content, it selects four pieces at random to download from peers that *have* those particular pieces. Once a peer has those four pieces, it shifts to a second phase of the algorithm which is based on a *rarest piece first* policy. Here, each piece's count is incremented based on HAVE and BITFIELD messages. The piece with the lowest count (but not zero) is selected as the next piece. The side effect of this policy is a peer will not get ahead of others in the download process. So, as we will show in Section 4, peers tend to complete at about the same time except for a few stragglers under uniform network transfer rates.

3 SIMULATION MODEL DESIGN AND IMPLEMENTATION

Our model of the BitTorrent protocol is written on top of ROSS (Carothers et al. 2000), which is an optimistically synchronized parallel simulator based on the Time Warp protocol (Jefferson 1985) written in the ANSI C programming language. In this modeling framework, simulation objects, such as a peer, are realized as a logical process (LP) and exchange time-stamped event messages in order to communicate. Each

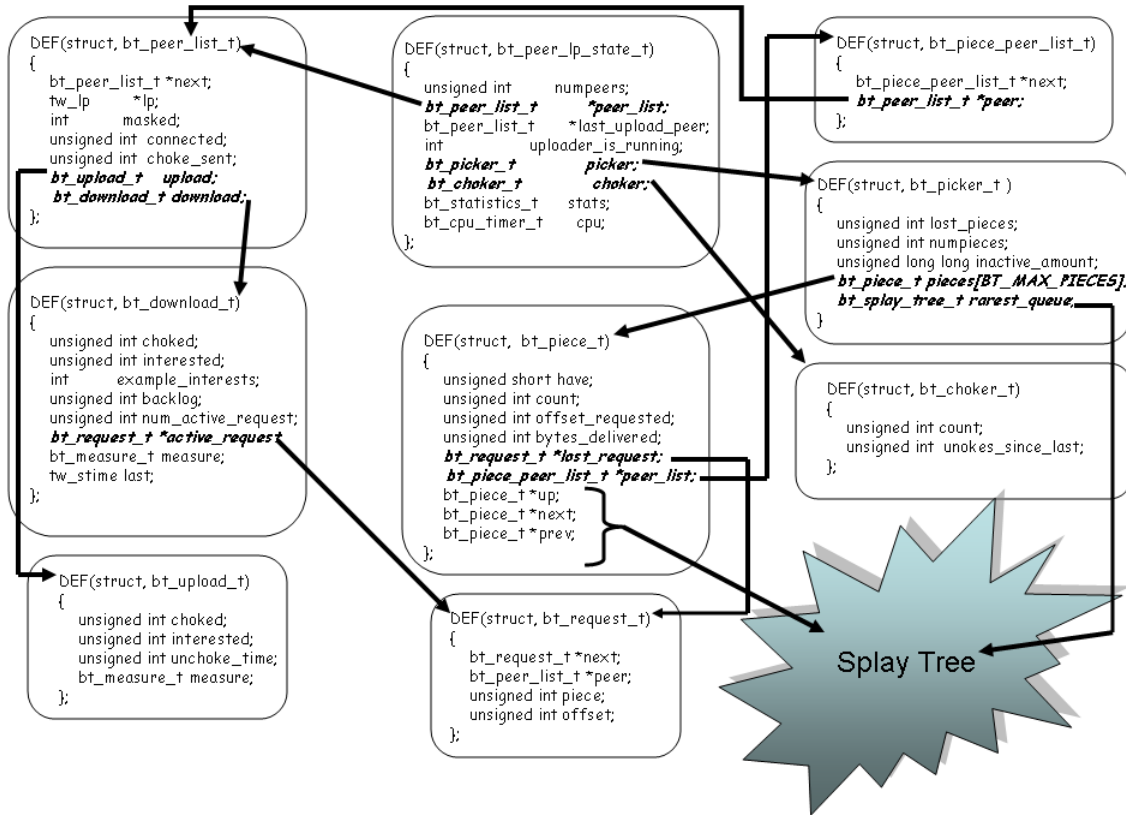


Figure 1: BitTorrent Model Data Structure

message within the BitTorrent protocol is realized as a time-stamped event message. Time stamps are generated based on computed delays through a synthetic network which realistically approximates today's home broadband service.

3.1 BitTorrent Model Data Structure

The data structure layout for our BitTorrent model is shown in Figure 1. At the core of the model is the peer state, denoted by `bt_peer_lp_state_t`. Inside each peer, there are three core components. First, is the `peer_list`, followed by the `picker` and the `choker`. The peer list captures all the upload and download state for each peer connection, as denoted by the `bt_peer_list_t` structure. A peer list can be up to 80 in length. The `picker` contains all the necessary state for a peer's piece picker. The `choker` manages all the required data for a peer's choker algorithm. This algorithm makes extensive use of the download data structure for each peer connection. Finally, each peer contains data structures to manage statistical information as well as simulated CPU usage that are used in protocol analysis.

Next, each peer connection has upload and download state associated with it. The `upload` denoted by the `bt_upload_t` request to for this particular piece.

`t` structure contains upload side status flags, such as `choked`, `interested`, etc. The download state denoted by the `bt_download_t`, is significantly more complex from a modeling perspective. In particular, this structure contains a list of active requests made by the owning peer. The `bt_request_t` data structure contains a pointer to the destination peer in the peer list along with the piece and offset information. Recall, that each 256 KB piece is further subdivided into partial chunks of 16 KB each. The offset indicates which 16 KB chunk this request is for within an overarching piece.

Now, inside the piece picker data structure, denoted by `bt_picker_t` is an array of piece structures along with a rarest piece priority queue. Inside each piece array element, denoted by the `bt_piece_t` is the current download status of that particular piece. Two key lists inside of the data structure are the `lost_request` and `peer_list`. The `lost_request` is a queue for requests that need to be re-made because the connection to the original destination peer was closed/terminated as per the BitTorrent protocol. The `peer_list` is the list of peers that *have* this particular piece (determined by receipt of a `HAVE` message). This list is used by the piece picker algorithm to select which peer to send the

We observe here that this `piece_peer` list is different from the previous ones in that it points to a container structure, `bt_piece_peer_t` which is just a list with a peer list pointer contained within it. It is this data design that results in significant memory savings over a static allocation of peer arrays. This enables us to manage our own piece-peer memory and reuse memory once a piece has been fully obtained. Similarly, we also manage `bt_request_t` memory. As a leecher-peer becomes a seeder-peer, it no-longer issues piece download requests. Thus, those memory buffers can be reused within the simulation model for other download requests and enables greater scalability in terms of the number of peer-clients that can be modeled.

The final key data structure within the piece picker is a Splay Tree priority queue (Ronngren and Ayani 1997) which is used to keep the rarest piece at the top of the priority queue. Our selection of this data structure over others such as a Calendar Queue (Brown 1988), is because of its low memory and high-performance for small queue lengths (i.e., less than 100). The key sorting criteria for this queue is based on counts of peers that have each piece. The lowest count piece will be at the top of the queue. Each peer LP manages its own rarest piece priority queue.

3.2 Tracker and Network Design

Our current protocol model creates a peer-set for each peer at the start of the simulation. Here, each peer LP is given a randomly generated set of peers. The peers then establish connections through our simulated home broadband network. Once connected, peers currently do not attempt to obtain subsequent or additional peer sets via a tracker. While this aspect is not realistic in terms of how BitTorrent operates, it does enable us to better understand the pure protocol dynamics without the outside influence of the tracker selection protocol. Here, our analysis approach is to decouple the client algorithms from the tracker selection strategies and study them independently. We are currently implementing a variety of locality preserving peer selection algorithms, such as those presented in (Bindal et al. 2006), to better understand how they perform under large “swarm” scenarios.

For our network design, we assume that for any given torrent, the amount of traffic generated, even if extremely large (100K peers or more), pales in comparison to the background traffic generated by other torrents, as well as HTTP and e-mail flows. Consequently, it makes little sense to model packet-level network dynamics of BitTorrent without considering the packet-level network dynamics of the much larger background network traffic. Moreover, BitTorrent operates at the level of a large payload size (16 KB) and collects bandwidth performance measures at time-scales much longer than the round-trip-time of any single TCP connection. So, while transient network behaviors do impact BitTorrent, they are amortized out as either increases or decreases in overall upload/download

capacity. Consequently, modeling BitTorrent at the packet-level as well as considering the complexity of TCP’s packet-level behavior, does not appear to be necessary. Moreover, without a reasonable background traffic packet-level model, a detailed packet-model’s behavior could not be accurately validated and is effectively “over-kill” for our modeling goals here. Our approach is to model available bandwidth to a particular peer and then sub-divide that bandwidth to its various upload and download streams. The details of this approach are beyond the scope of this paper.

3.3 Model Tuning Parameters

In terms of tuning parameters, BitTorrent has on the order of 20 or more which are beyond full consideration here. However, we do focus on two key parameters which have a profound impact on simulator performance. The first is `max_allow_in`. This parameter determines the maximum number of peers a peer will make connections to or accept requests from. This parameter determines how long a peer’s `peer_list` is which impacts the complexity of the piece picker and choker algorithms. Another key parameter is `max_backlog` which sets a threshold on the number of outstanding requests that can be made on any single peer connection. In the BitTorrent implementation, `max_backlog` is hard coded to be 50 unless the data transfer rate is extremely high (i.e., > 3 MB/sec), in which case it can go beyond that value. So, an approximate upper bound on the number of request events that can be scheduled is the product of `max_allow_in`, `max_backlog`, and the number of peers. A consequence of this product is that memory usage in the simulation model grows very quickly as we will observe in the next section..

4 EXPERIMENTAL RESULTS

All experiments were conducted on a 16 processor, 2.6 GHz Opteron system with 64 GB of RAM running Novell SuSe 10.1 Linux. For this performance study, all experiments are conducted on a single processor as the parallel version of the model is still under development.

4.1 Model Validation

In order to validate our BitTorrent model, we created three tests: (i) *download completion test*, (ii) *download time test*, and (iii) *message count test*. While there is no consensus in the BitTorrent community on a valid BitTorrent implementation because of variability that is acceptable within the protocol, we believe these tests provide us with some confidence in the behavioral accuracy of our model.

The first test asks the most basic question, *did all leecher peers obtain a complete copy of the file?*. To conduct this test, we executed our model in 16 different configurations based

Table 1: Number of Message Received per Type per Simulation Scenario

scenario	choke	unchoke	interested	not interested	have	request
128 peer, 128 pieces	10402	10402	30693	26539	1247294	333719
256 peer, 128 pieces	21181	21181	66213	57520	2552040	669542
512 peer, 128 pieces	42964	42964	144466	122469	5129973	1319830
1K peer, 128 pieces	86240	86240	287397	240759	10271737	2668919
128 peer, 256 pieces	10584	10584	42933	38899	2494655	622895
256 peer, 256 pieces	21399	21399	96786	86870	5104101	1236486
512 peer, 256 pieces	43013	43013	208328	186197	10258933	2517261
1K peer, 256 pieces	85753	85753	389975	348770	20543479	5028309
128 peer, 512 pieces	10950	10950	68810	63851	4989376	1177809
256 peer, 512 pieces	21661	21661	137811	128039	10208231	2350252
512 peer, 512 pieces	43258	43258	294295	271613	20517877	4706605
1K peer, 512 pieces	86051	86051	581193	537537	41087991	9521465
128 peer, 1K pieces	11240	11240	104061	99097	9978815	2258058
256 peer, 1K pieces	21979	21979	206557	196053	20416487	4531166
512 peer, 1K pieces	43340	43340	396434	373540	41033714	9000343
1K peer, 1K pieces	10402	10402	30693	26539	82178039	18316898

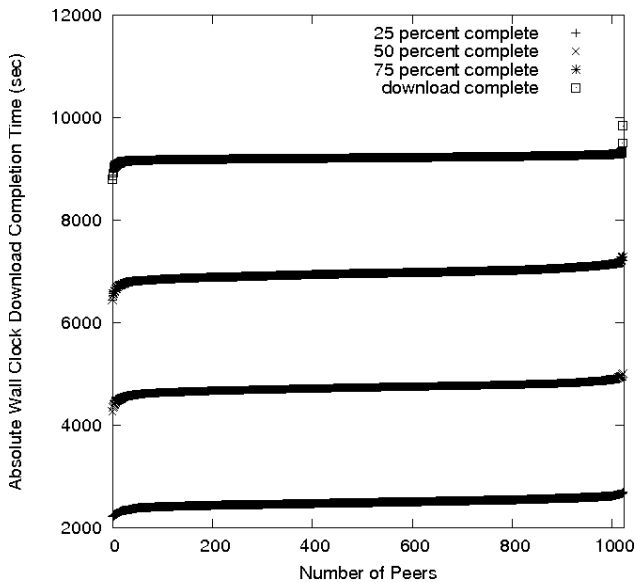


Figure 2: Simulated Download Completion Times (seconds) for 1,024 Peer, 1,024 Piece Scenario

on the number of peers and number of pieces. The number of peers ranged from 128 to 1,024 by a power of two. Similarly, the number of pieces also ranged from 128 to 1,024 by a power of two. At the end of each simulation run, we collected statistics on each peer. In particular, we noted how many remaining pieces a peer had, which was zero in all cases. Furthermore, we observed that all pending requests should have been satisfied. Thus, the active request list for each connection should be empty. We confirmed for all 16 cases that no requests were pending and in fact all request memory buffers

had been returned to the request memory pool, thus ensuring no memory leaks existed. Finally, we also ensured that as a piece is downloaded we correctly free the `peer_list` that have that piece and remove it from our rarest piece priority queue. Again, this verifies we do not have any memory leaks in the management of the piece-peer list structures and serves as a cross-check that all pieces have been correctly obtained by a peer.

In the second test, we want to know the distribution in time of when a peer completes the download of the file. We then verify the shape of our download times against those most recently published in (Bindal et al. 2006). For this test, we use the 1K peer, 1K piece scenario. We observe that at each milestone, 25%, 50%, 75%, and 100% (download complete), most of the peers reach it at the same point in time, as shown in Figure 2. This trend is attributed to the rarest piece first policy used to govern piece selection coupled with fair *tit-for-tat* trading. This prevents for the most part any peer “getting ahead” in the overall download process. We do however note, there does appear to be some “early winners” and “late losers” in the process. This phenomenon occurs because not all peer sets have access to all pieces at the same time. Some peer sets are losers (i.e., many hops away from the original seeder) and peer sets contain the seeder enabling them to get to the rarest pieces more readily. The shape of our completion download time curve is confirmed by the emulated results presented in (Bindal et al. 2006). Additionally, we find the real measurement data in (Legout et al. 2005) reports a similar download time distribution shape. However, a key difference is that the variance is much greater leading not to a flat line as we have, but a positive slope line. We attribute this difference to the measurement data covering only an extremely small “swarm” (only 30 leechers with 9 peers). Thus, the network paral-

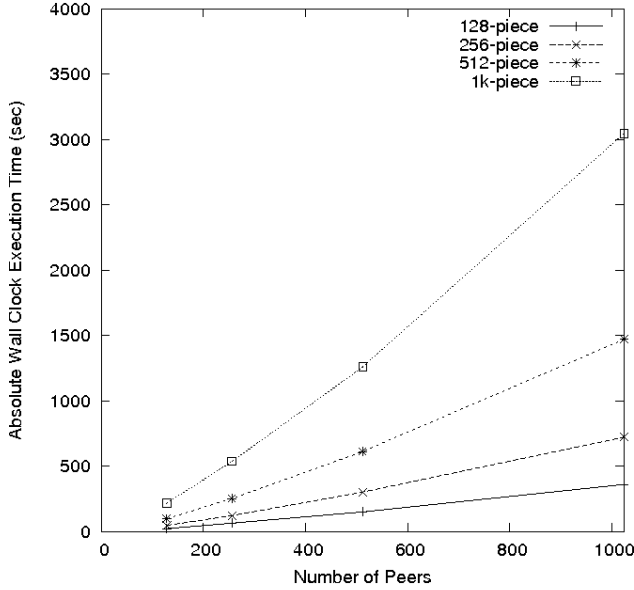


Figure 3: Model Execution Time as Function of Number of Pieces and Number of Peers

lelism is not available because of fewer connections. Therefore, downloads will be more serialized yielding longer, more staggered download completion times.

In the last test, we validate our message count data as shown in Table 1 against the real measurement data reported in (Legout et al. 2005). There are two key trends that appear to point to proper BitTorrent operation. The first is that the number of *choke* and *unchoke* messages should be equal. In all 16 configurations we find this assertion to be true. This is because the choke algorithm forces these messages to operate in pairs. That is for every peer that is unchoked, there will be a following choke message in the future until the file download is complete. Second, the number of *interested* messages should be slightly higher but almost equal to the number of *not interested*. We observe this phenomenon across all 16 model configurations. Finally, we observe that the number of *have* and *request* messages meet expectation. In the case of *have* messages, they are approximated by the number of peers times the number of pieces times the number of peer connections per peer. In the case of the 1K peer, 1K piece scenario, this is bounded by $80 * 1,024 * 1,024 = 83,886,080$. Likewise, the number of requests has a lower bound of number of pieces times 16 chunks per piece times the number of peers. The reason this is a lower bound is because of endgame mode, which allows for the same piece/offset to be requested many times across different peers.

4.2 Model Performance

To better understand how our BitTorrent model scales and affects simulator performance, we conducted the following se-

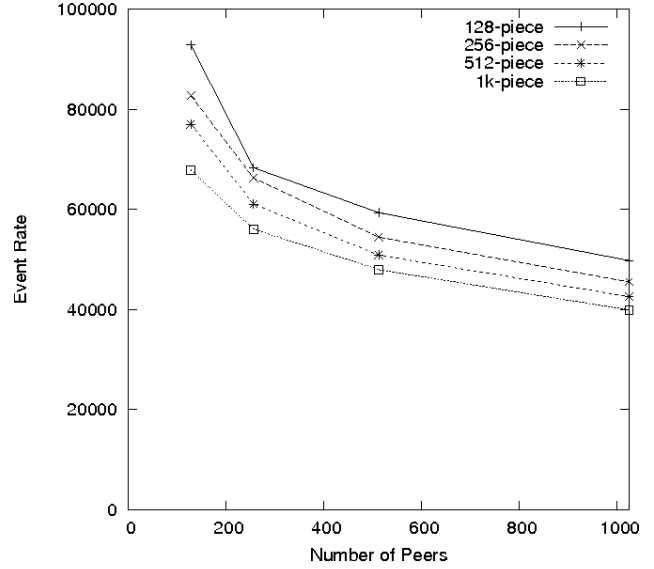


Figure 4: Model Event Rate as Function of Number of Pieces and Number of Peers

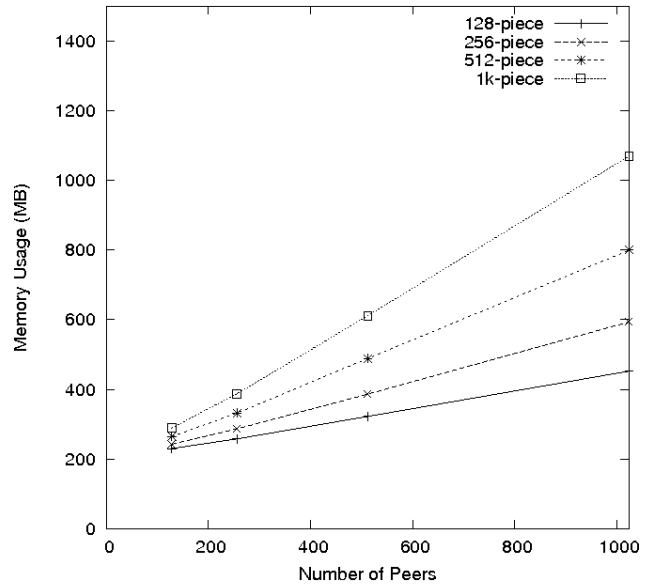


Figure 5: Model Memory Usage in MB as Function of the Number of Pieces and the Number of Peers

ries of experiments. In the first set shown in Figure 3, we plot the simulation execution time as a function of the number of peers and the number of pieces. The number of peers and pieces range from 128 to 1,024 by a power of 2 yielding 4 sets of 4 data points each. We observe that by increasing the number of peers for small (128) piece files does not impact simulator performance significantly. However, as the number of pieces grows, the slope of the execution time increases tremendously as the number of peers increase. We attribute this behavior to the increased complexity in the download side

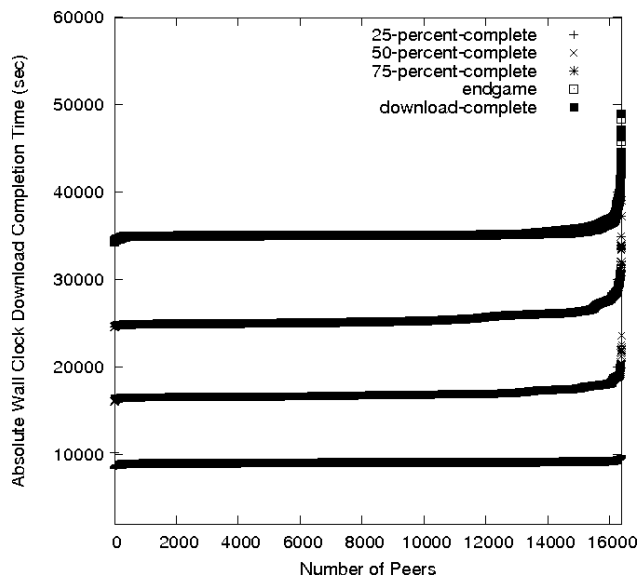


Figure 6: Simulated Download Completion Times (seconds) for 16,384 Peer, 4,192 Piece Scenario (this simulation run required 15.14 GB of RAM and 59.66 hours to execute and an event rate of 35,179)

of a peer connection as a consequence of a large number of pieces to consider. Additionally, by increasing the number of peers and pieces the overall event population increases which leads to larger event list management overheads. To verify this view, we plot the event rate as a function of the number of peers and pieces in Figure 4. We observe that the event rate is highest (close to 100K events/second) when both peers and pieces are small in number (i.e., 128, 128 case). However, in the 1K peer, 1K piece case, we observe that the event rate has decreased to only 40K events/second because of more work per event as well as higher event list management overheads.

The memory usage across the number of peers and pieces is shown in Figure 5. Here, we observe that memory usage grows much slower for smaller piece torrents than for the larger. For example, the 1k peer, 128 pieces scenario only consumes 452 MB of RAM or 441 KB per peer whereas the 128 peer, 1k pieces scenario consumes 289 MB or 2.3 MB per peer. This change in memory consumption is attributed to two reasons. First, as the number of pieces grow, the availability of pieces to select from grows as well. This in turn allows more requests to be simultaneously scheduled which results in a larger peak event population and increases the overall event memory required to execute the simulation model. Increasing the number of peers has a similar impact in that it will also increase the event population (and demand on request memory buffers) which raises the amount of memory necessary to execute the simulation model.

In the last performance curve, we show the download completion times for a very large 16K peers, 4K pieces scenario. We note here that we observe a larger population of “late”

downloaders at each milestone. Observe that endgame and download completion occur extremely close to each other. Thus, peers do not spend a great deal of time in endgame mode overall. For simulator performance, this model consumed 15.14 GB of RAM and required almost 60 hours to complete. At first pass 60 hours does appear to be a significant amount of time, but we observe that it is orders of magnitude smaller than the measurement studies that require many weeks or even months of peer/torrent data collection.

Finally, we report the completion of a simulation scenario with 128,000 peers, 128 pieces, 32 connections per peer, and a `max_backlog` of eight. The impact on memory usage was significant. This scenario only consumes 8.15 GB of RAM or 67 KB per peer which points to how the inter-play between pieces, peers and requests dramatically effects the underlying memory demands of the simulation model.

5 CONCLUSIONS

In this paper, we demonstrated a lightweight, memory efficient BitTorrent model that enables researchers to investigate large-scale “swarm” scenarios. The key result from this case study is that we are able to execute a detailed BitTorrent model consuming only 67 KB to 2.3 MB per peer which is significantly less than the operational BitTorrent software consumes.

In the future, we plan to examine ways to further reduce the execution time and enable scaling to swarm sizes of 100’s of thousands by employing commodity multi-processor hardware. However, a key challenge here is how to properly synchronize the BitTorrent model such that events are processed in time-stamp order? Because of both the dynamic state and communication pattern (i.e., random) complexity, neither a strict conservative (i.e., blocking) or optimistic (i.e., speculative) synchronization protocol is ideal. We believe some sort of hybrid protocol will need to be devised.

REFERENCES

- Abilene/Internet II Usage Policy.
<http://abilene.internet2.edu/policies/cou.html>
- A. R. Bharambe, C. Herley, and V. N. Padamanbhan, “Analyzing and Improving BitTorrent Performance”, Microsoft Research Technical Report, MSR-TR-2005-03, February, 2005.
- R. Bindal, P. Cao, W. Chan, J. Medval, G. Suwala, T. Bates and A. Zhang. “Improving Traffic Locality in BitTorrent via Biased Neighbor Selection”, In *Proceedings of the 2006 International Conference on Distributed Computing Systems*, July 2006, Spain.
- BitTorrent Source Code, ver. 4.4.0, Linux Release.
www.bittorrent.com/download.myt.

- BitTorrent, News Release: Partnership with Warner Brothers, 2006. www.bittorrent.com/2006-05-09-Warner-Bros.html.
- R. Brown, "Calendar queues: A fast $o(1)$ priority queue implementation for the simulation event set problem," *Communications of the ACM (CACM)*, vol. 31, pp. 1220–1227, 1988.
- C. D. Carothers, D. Bauer and S. Pearce. "ROSS: A High-Performance, Low Memory, Modular Time Warp System", In *Proceedings of the 14th Workshop of Parallel on Distributed Simulation (PADS 2000)*, pages 53–60, May 2000.
- C. D. Carothers, D. Bauer and S. Pearce. "ROSS: Rensselaer's Optimistic Simulation System User's Guide". Technical Report #02-12, Department of Computer Science, Rensselaer Polytechnic Institute, 2002, <http://www.cs.rpi.edu/tr/02-12.pdf>.
- J. Cowie, A. Ogielski and B.J. Premore. "Internet Worms and Global Routing Instabilities" In *Proceedings of the Annual SPIE 2002 Conference*, July 2002.
- Z. Ge, D. R. Figueredo, S. Jaswal, J. Jurose, D. Towsley, "Modeling Peer-Peer File-Sharing Systems", In *Proceeding of the IEEE INFCOM 2003*.
- C. Gkantsidis and P. Rodriguez, "Network Coding for Large Scale Content Distribution", In *Proceedings of IEEE INFOCOM 2005*, Miami. March 2005.
- P. Grant and J. Drucker. "Phone, Cable Firms Rein In Consumers' Internet Use Big Operators See Threat To Service as Web Calls, Videos Clog Up Networks", *The Wall Street Journal*, October 21, 2005, page A1.
- D. R. Jefferson. "Virtual time". *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.
- A. Legout, G. U-Keller, and P. Michiardi. "Understanding BitTorrent: An Experimental Prospective", INRIA Technical Report, #00000156, Version 3, November 2005.
- R. LeMay, "BitTorrent creator slams Microsoft's methods", June 21, 2005. ZDNet Australia. <http://www.zdnet.com.au/news/software/0,2000061733,39198116,00.htm>
- NS: Network Simulator, 2006. <http://www.isi.edu/nsnam/ns/ns.html>
- A. Parker, "The True Picture of Peer-to-Peer File-Sharing", <http://www.cachelogic.com/research/slidel.php>.
- A. Parker, "P2P in 2005", http://www.cachelogic.com/research/2005_slide01.php
- L. Peterson, T. Anderson, D. Culler, and T. Roscoe. "A Blueprint for Introducing Disruptive Technology into the Internet". In *Proceedings of the First Workshop on Hot Topics in Networking (HotNets-I)*, October, 2002.
- Planet Acceptable Use Policy. 2006. <http://www.planet-lab.org/php/aup/>
- J. A. Pouwelse, P. Garbacki, D. H. J. Epema, and H. J. Sips, "The BitTorrent P2P File-Sharing System: Measurements and Analysis", In *Proceedings of the 4th International Workshop on Peer-2-Peer System (IPTPS '05)*, February 2005.
- R. Ronngren and Rassul Ayani, "A comparative study of parallel and sequential priority queue algorithms," *ACM Transactions on Modeling and Computer Simulation*, vol. 7, no. 2, pp. 157–209, April 1997.
- R. Shaw. "BitTorrent users, ignore Opera at your inconvenience". *ZDNet Blogs*, February, 17th, 2006. <http://blogs.zdnet.com/ip-telephony/?p=918>
- Slyck – Home Page, 2006. <http://slyck.com/bt.php?page=21>

AUTHOR BIOGRAPHIES

CHRISTOPHER D. CAROTHERS is an Associate Professor in the Computer Science Department at Rensselaer Polytechnic Institute. He received the Ph.D., M.S., and B.S. in Computer Science from Georgia Institute of Technology in 1997, 1996, and 1991, respectively. His research interests include parallel and distributed systems, simulation, networking, and computer architecture.

RYAN LAFORTUNE is a Ph.D. student in the Computer Science Department at Rensselaer Polytechnic Institute. His research interest is in cover sets for distributed systems.

WILLIAM D. SMITH is Senior Technical Staff at the General Electrical Global Research Center. He received the M.E. and B.E. in Electrical Engineering from Rensselaer Polytechnic Institute in 1984 and 1982 respectively. His current research interest are in computer architecture, digital rights management systems and content distribution networks.

MARK GILDER is Senior Technical Staff at the General Electrical Global Research Center. He received the Ph.D., M.S. and B.S. in Computer Science from Rensselaer Polytechnic Institute in 1994, 1990 and 1988 respectively. His current research interest are in static code analysis for large-scale scientific computations, digital rights management and content distribution networks.