BitStore: An Incentive-Compatible Solution for Blocked Downloads in BitTorrent

Anirudh Ramachandran, Atish Das Sarma, and Nick Feamster
College of Computing, Georgia Tech
{avr, atish, feamster}@cc.gatech.edu

ABSTRACT

As many as 30% of all files shared on public BitTorrent networks suffer from the lack of "seeders" (peers that have complete copies of the file being shared); peers attempting to download such a file ("leechers") may have to wait indefinitely to obtain certain file chunks that are not distributed in the file's network of peers (the "swarm"). We call this the *Blocked Leecher Problem* (BLP).

To alleviate BLP, we propose *BitStore*, a larger, secure network of BitTorrent users (not necessarily all sharing the same content) where nodes offer their resources (such as disk space and bandwidth) for public use. Peers sharing any file can use the storage network to maintain replicas for each chunk of the file. Any leecher seeking chunks that are absent from in its own swarm can query the public network, locate the node storing the said chunks, and retrieve them. *BitStore* also provides robust incentives for nodes contributing resources: In return for storing and serving chunks, such nodes can negotiate *micropayments* using a second-price auction. Peers who receive these credits may later use them to retrieve blocks *they* need from the storage network.

This paper quantifies the BLP, presents an overview of the *BitStore* design, and discusses various challenges related to storage management and incentives.

1. Introduction

BitTorrent [13] is one of the most common content-distribution protocols on the Internet: Estimates place its contribution to *all* Internet traffic at 35% [1]. Although BitTorrent works well for popular content, it frustrates downloaders who wish to obtain less popular content, who may often have to wait in the swarm indefinitely to meet peers that have the blocks they need. This suboptimal behavior in BitTorrent, which we call the *Blocked Leecher Problem (BLP)*, is common: Our investigation of torrent listings from a torrent aggregator, Mininova [3], shows that almost 30% of torrent file swarms analyzed have zero seeders but one or more leechers.

We briefly describe BitTorrent to explain why BLP is common. A BitTorrent "swarm" comprises seeders (peers who have complete copies of the file), leechers (peers who have partial copies of the files), and a tracker (a per-torrent server that enables peers to discover each other). Files are divided into chunks (usually 64 – 512 KB each), and peers distribute chunks among each other. A seeder only uploads content, while leechers upload and download content until their downloads complete (at which point they become seeders). In a swarm with no seeders, leechers' downloads will only complete if the entire file can be reconstructed from chunks that leechers in the swarm are sharing. If a swarm does not collectively possess all chunks of the file, leechers will never be able to recover more than a fraction of the file (after sharing the chunks they possess among themselves); they will then wait indefinitely for blocks that no leechers in the swarm have.[1] BLP is not to be confused with the well-known *last-block problem*, which is merely an instance of the coupon collector problem; BLP occurs when some chunks of the file are not shared by *any* peer in the swarm.[2] If the overlap between the active periods of leechers and the last seeder are insufficient to distribute the whole file, the leechers that remain in the swarm (as well as leechers that arrive subsequently) find themselves facing BLP.

To solve BLP, we propose *BitStore*, which augments the BitTorrent protocol by providing a persistent global store that is shared among swarms. For each swarm, peers agree to lazily maintain a minimum number of replicas of every chunk of the original file. Essentially, the distributed storage mechanism offered by *BitStore* supplants the need for a seeder, since all blocks of the original file are guaranteed to reside in *BitStore*. Unfortunately, the clients belonging to a particular swarm are not sufficient to maintain these replicas (otherwise, they would never face BLP as they would be able to retrieve the chunks they need). Instead, *BitStore* creates an auxiliary storage network, $\mathcal{S}$, comprising clients from *other swarms* such that the total number of nodes in $\mathcal{S}$ is large (in particular, much greater than the number of clients in any single swarm). In practice, this network may include all peers tracked by an organization (*e.g.*, a torrent aggregator). Since even a single popular torrent aggregator may track many millions of active peers, the size of $\mathcal{S}$ could be quite large [4].

In *BitStore*, leechers can retrieve missing file chunks from $\mathcal{S}$, since *seeders would have replicated all chunks of the file*

[1]Usually, if a seeder leaves a swarm, it is unlikely to return and contribute content; he has no incentive to do so. A swarm with no seeders can be thus "blocked" until altruistic seeders return (usually after out-of-band pleas for help, illustrated by the numerous plaintive 'Please seed!' messages on torrent tracker websites.)

[2]Because BitTorrent peers use a Local Rarest First policy for picking new blocks to download, BitTorrent is less prone to the last block problem than most P2P filesharing networks [8].

on $\mathcal{S}$ before leaving. In return for hosting content for the storage network $\mathcal{S}$, nodes obtain *micropayments* [23], where the amount of payment received is determined according to a second-price auction. A node can then later use these tokens to retrieve chunks from $\mathcal{S}$ for other files on which it is blocked.

This paper's main contribution is the design of *BitStore*, which is, to our knowledge, the first system to address the Blocked Leecher Problem. *BitStore* uses the spare bandwidth and disk-space of peers to create a unified, quasi-persistent store for blocks of files for all participating swarms. Despite its conceptual simplicity, *BitStore* must address several challenges that arise from the fact that the distributed store, $\mathcal{S}$, is maintained by BitTorrent peers themselves, which must be properly incented to donate storage and bandwidth resources to $\mathcal{S}$. *BitStore*'s design provides these incentives and has the following salient features:

- *BitStore*'s lazy replication scheme guarantees that, with high probability, the distributed store, $\mathcal{S}$, will contain the complete file chunks for unpopular files that lack seeders (Section 4.2).
- *BitStore*'s distributed store is robust to resource exhaustion attacks (Section 4.3).
- *BitStore* provides strong privacy and anonymity guarantees. In particular, peers that store blocks for $\mathcal{S}$ have no knowledge of the blocks that they are storing for other peers (Section 4.4).
- *BitStore* provides incentives (through token-based micropayments) for peers to contribute resources to $\mathcal{S}$, which guarantees that $\mathcal{S}$ (Section 5).

In addition, *BitStore* operates in parallel with BitTorrent's chunk swapping mechanisms and thus requires no modifications to existing BitTorrent protocols or software.

After presenting a brief overview of BitTorrent and related work in Section 2, we present an overview of the *BitStore* protocol in Section 3. Sections 4 and 5 describe the details of storage management and incentives, respectively; we conclude in Section 6.

2. Background and Related Work

We describe the BitTorrent protocol and its desirable properties, as well as why BLP is common. (Section 2.1). Next, we present relevant related work (Section 2.2).

2.1 BitTorrent Overview

Content is published using a `.torrent` file, a small (typically less than 1 MB) file that contains hashes of all chunks of the content being distributed, as well as the domain name and port of the tracker. A single machine typically tracks multiple torrents, and busy trackers often track thousands of torrents (and, hence, hundreds of thousands of clients). The majority of public trackers are offered as a free service to users, by a small set of organizations (*e.g.*, Mininova [3], The Pirate Bay [5], Demonoid [2], etc.); these organizations also offer web-based interfaces for users to search, locate, and obtain torrent files to start their downloads. We refer the reader to Cohen's original paper [13] and subsequent research [18, 25, 8] for more details.

BitTorrent has many desirable properties, including: (1) *Scalability:* A BitTorrent network's capacity to serve content grows with more clients ("self scalable"); (2) *Robust Incentives:* BitTorrent offers participants robust incentives to contribute content to the swarm; and (3) *Avoidance of the Last-Block Problem:* BitTorrent participants employ the Local Rarest First (LRF) policy in choosing blocks to download from peers, diminishing the effects of the last-block problem. Unfortunately, BitTorrent does not provide any incentives for seeders to stay to contribute blocks after their downloads have completed, which can give rise to BLP, especially for unpopular files.

2.2 Related Work

We briefly survey previous work on replicated systems and incentive systems for peer-to-peer networks.

Replicated Systems. Replicated systems have been popularly used for fault tolerance including byzantine fault tolerant [11] systems such as BAR [6] and quorum systems [22]. *BitStore* does not provide byzantine fault tolerant guarantees, but instead uses cryptographic techniques to verify chunks served by nodes. *BitStore*'s storage infrastructure is based on the DHash [14] block storage system.

Incentives in P2P Networks. In contrast to barter systems like BitTorrent [13], filesharing networks such as Kazaa [19] and eMule [15] offer reputation-based incentive mechanisms to reward altruistic peers. Reputation systems(*e.g.*, [20]) are susceptible to collusion and are not scalable; *BitStore* instead uses pairwise currency exchanges to incent peers providing service. Incentive mechanisms have also been proposed for new [21] P2P content-distribution systems, media streaming [17], and for the general case [16]. Recent research has showed how BitTorrent's incentive system may be exploited by selfish peers to gain better performance [24].

3. The BLP and BitStore

We first describe the Blocked Leecher Problem (Section 3.1) and discuss its prevalence in today's BitTorrent networks. We then present an overview of *BitStore* (Section 3.2).

3.1 The Blocked Leecher Problem

The Blocked Leecher Problem (BLP) affects any peer-to-peer network where files are divided into chunks and stored across peers (BitTorrent is one example of such a network). As a file decreases in popularity, the number of peers distributing chunks from that file also decreases, as does the likelihood that the set of peers will collectively have all chunks of the file.

The last seeder to leave a swarm might have left when there were few leechers, but as more leechers join later, all of them face BLP. Note, in particular, that while the last seeder

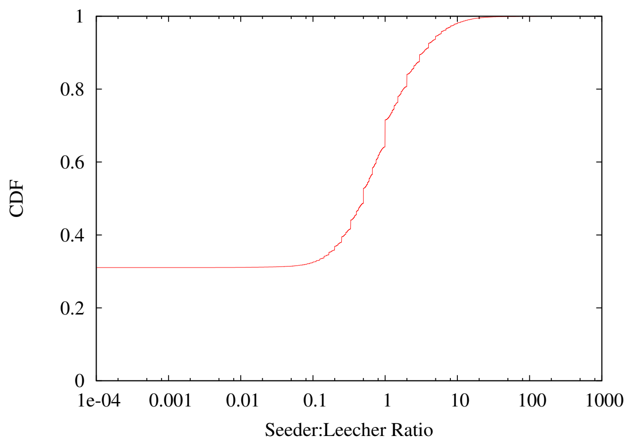

Figure 1: CDF of the ratio for seeders to leechers for a snapshot of 138,500 torrents from the Mininova torrent aggregator.

was active, its upload bandwidth was likely *not* saturated by peer downloads; the BLP occurs because the overlap window between the active periods of the seeder and leechers are small (or zero, for leechers who join after the seeder leaves.) Figure 1 shows that the potential for the BLP is high: 30% of all torrents have at least one leecher with no seeders. Leechers in these torrents have little chance of completing their downloads.

3.2 BitStore Overview

BitStore exploits the following property of BitTorrent swarms: Though the number of peers in any single swarm may fluctuate, the *total* number of peers at any time is large. *BitStore* aggregates these peers into a single loosely organized network (which we call $\mathcal{S}$, for "storage" network), where each peer may use other peers as a "buffer" for content that cannot be distributed in its own swarm.

$\mathcal{S}$ is the union of all peers that are tracked by a cooperating set of trackers (*e.g.*, the set of all trackers run by a single organization). The peers in $\mathcal{S}$ maintain a fixed number of replicas for all chunks *in their respective swarms* using $\mathcal{S}$ as a buffer. If a node discovers a shortage of copies of any particular chunk in its swarm, it stores that chunk to $\mathcal{S}$ to allow other nodes in its swarm to retrieve them later.

To encourage peers to participate in $\mathcal{S}$, nodes that serve chunks from $\mathcal{S}$ receive payment (in "tokens") from other nodes that retrieve blocks that they are serving (the price of each block is dictated by market mechanism that we discuss in detail in Section 5). The coordinating trackers act as a Trusted Third Party for money exchanges: they possess cryptographic key pairs, can convert real-world money to tokens and vice versa, and can sign and verify messages on behalf of nodes connected to them.

Nodes in *BitStore* can act in three capacities: 1) *Departing seeders*, who use $\mathcal{S}$ to maintain replicas for file chunks; 2) *Buyers*, who download file chunks stored by nodes in $\mathcal{S}$; and 3) *Sellers*, who offer resources (bandwidth, disk space) for $\mathcal{S}$ to buyers. Figure 2 describes at a high-level the protocol a node follows for each of these roles.

```
CASE 1: NODE $A$ WISHES TO DISTRIBUTE REPLICAS OF CHUNK $C$
        IN $\mathcal{S}$:
        Let $s$ = set of nodes in $\mathcal{S}$ designated to store $C$
        foreach node $n$ in $s$
            if $n$ does not have a copy of $C$
                replicate $C$ at $n$
CASE 2: NODE $A$ WISHES TO DOWNLOAD A COPY OF CHUNK $C$
        FROM $\mathcal{S}$:
        Let $s$ = set of nodes in $\mathcal{S}$ designated to store $C$
        Let $q = |r|$, where $r \subseteq s$ are nodes that still have a copy of $C$
        Choose a node $n \in r$ at random
        Participate in a second-price auction at $n$ for $C$
        if $A$ is the winning bidder
            securely pay $n$ the second-highest price (in tokens)
            in return for $C$
        else
            Back off, and repeat CASE 2 possibly after earning
            more tokens
        if $|r| < |s|$
            Execute CASE 1 to replenish replicas of $C$ in $\mathcal{S}$
CASE 3: NODE $A \in \mathcal{S}$ HAS A SET OF CHUNKS, $C$,
        FOR SALE
    Let $b$ = upload bandwidth $A$ can spare for service
    Let $N_c$ = number of chunks $A$ is willing to serve at any instant
    Let $t = \frac{N_c}{b}$, the period between auctions at $A$
    every $t$ units of time
        Aggregate all chunks that had requests in the last $t$ units
        into the set $C_r = \{c_1, c_2, \ldots, c_k\}$, $C_r \subseteq C$
        foreach $c_i \in C_r$
            Let $p_i$ = reserve price for chunk $c_i$
            if number of requesters for $c_i > 1$
                Conduct second-price auction among requesters of $c_i$
                if winning bid meets $p_i$
                    Accept payment and serve $c_i$ to the winner
            else
                if requester's price meets $p_i$
                    Accept payment and serve $c_i$
            Refuse service to everyone else
```

Figure 2: The *BitStore* protocol for nodes assuming various roles.

4. Storage Management

This section presents the distributed replica storage system in *BitStore*. We first describe the basic storage mechanisms based on DHash and necessary modifications to DHash to facilitate replica maintenance. We then describe *BitStore*'s defenses against resource exhaustion and other guarantees regarding privacy and anonymity.

4.1 Basic System Structure

Peers in any swarm must be able to use $\mathcal{S}$ to store and retrieve blocks 1) within a reasonable number of lookups, 2) without any persistent bookkeeping information, and 3) in the presence of failures and changing network conditions. To achieve these properties, $\mathcal{S}$ relies on the DHash [14] Distributed Hash Table (DHT) block storage system, layered on top of Chord lookup protocol [28], to efficiently distribute chunks among nodes in $\mathcal{S}$. DHash offers simple primitives for operating on blocks (*i.e.*, file chunks) over nodes in the Chord ring: 1) `put(block)`: Computes block's key by hashing its contents, and sends the block to the appropriate server (as dictated by Chord rules) for storage; 2) `get(key)`: Locate the server responsible for storing the block identified by key and retrieve the requested block.

BitStore should be able to serve content even if nodes fail or otherwise unpredictably depart. Thus, *BitStore* must distribute content so that nodes that fail together (due to network topology or malice) will not affect *all* replicas of any given chunk. Chord offers provable guarantees for the number of lookups in locating a server storing a key, robustness to failures, and scalability, but *BitStore* requires several modifications to the basic DHash scheme to allow maintenance of multiple replicas, to defend against resource exhaustion, to protect privacy, and to provide incentives for nodes to contribute resources to $\mathcal{S}$.

4.2 Replica Maintenance

In Chord, every peer node, and each block, hashes to exactly one position, but *BitStore* must store multiple copies of each block in $\mathcal{S}$. Accordingly, *BitStore* computes Chord indices using a hash function of the form $h(chunk\_hash, secret, i)$, where $chunk\_hash$ refers to a hash that uniquely identifies the chunk; $secret$ is a secret that is provided by the tracker and is known only to nodes in the swarm the chunk belongs to; and i is the index of the replica. The index of the replica, i, runs from 0 through q_s—the maximum amount of replicas of a chunk that need to be stored on $\mathcal{S}$—implying that one block is stored at q_s distinct nodes in $\mathcal{S}$. Though the node performing the lookup does not have a copy of the chunk (and hence cannot directly compute the hash of the chunk), it may obtain the hash of the chunk from the `.torrent` file; thus, $chunk\_hash$ refers to the hash of the hash of the chunk.

$secret$ is a per-torrent secret provided by the tracker that allows members of a particular swarm to locate replicas for their file, but prevents nodes in $\mathcal{S}$ that are not in *that* swarm from discovering these replicas (*i.e.*, since only nodes in that swarm know the secret, other nodes cannot compute the Chord indices for these replicas).

Using estimates from a snapshot of a torrent aggregator's statistics, we claim that even modest contributions from nodes can achieve high replication factors (Appendix A provides the calculation for this estimate).

Claim 4.1 *For a uniform number of replicas for each swarm, say $q = 10$, any node in $\mathcal{S}$ will have to store only a few hundred MB of data.*

Modifications to DHash *BitStore* modifies DHash's access primitives and presents the following interface to clients:

- `mput(block)`: `mput` is a wrapper to the DHash `put` primitive, and encapsulates the algorithm in Case 1 of Figure 2. The node computes hash functions to generate q_s keys, locates the nodes in $\mathcal{S}$ responsible for the keys, and issues `store` requests to each of them.
- `mget(key)`: `mget` is the implementation for Case 2 of Figure 2, and is issued by a node wishing to obtain a chunk from $\mathcal{S}$. `mget` locates nodes as in `mput`, but the node ultimately buys the chunk from just one "seller" of the chunk (typically in an auction). Upon winning an auction, the node also issues an `mput` to bring the number of replicas up to q_s.

When a seeder arrives, it uses `mput` to create the initial set of q_s replicas in $\mathcal{S}$. Although this initial replica creation can tax the seeder's bandwidth [9], this overhead can be reduced with multicast [10] or a BitTorrent-like distribution mechanism. The swarm's tracker stores and informs any node in that swarm both the secret ($secret$) and the number of replicas (q_s); any node in the swarm can then compute the hash functions for any chunk and retrieve chunks directly from nodes in $\mathcal{S}$.

Replicas can be depleted if either a node storing a certain replica fails, or the replica's *lifetime* expires (after which the node can safely assume that no leecher is seeking this chunk). Replica maintenance occurs *lazily*: After a leecher issues an `mget` request to obtain a chunk, it issues an `mput` for the *same* chunk, which increases number of replicas of that chunk in $\mathcal{S}$ to q_s. Any node which has the chunk ignores the `mput` (*i.e.*, it does not store duplicates of the same chunk). Though chunks may be replicated less than q_s times, a sufficiently large value of q_s and uniformly distributed storage across $\mathcal{S}$ implies that, with high probability, at least one replica will be available.

With q_s replicas, the probability that no replica is available is exponentially small in q_s. Although, in the worst case, `mput` should check for successful insertion of all q_s replicas, the following claim shows that verifying the successful insertion of only $c < q_s$ replicas can provide the same asymptotic failure guarantees.

Claim 4.2 *A leecher can ensure replication at only c out of q_s nodes ($c < q_s$), where c is selected uniformly at random, without compromising on the asymptotic failure probability.*

Appendix B provides a proof sketch for this claim. To provide intuition on the failure probability we can guarantee, suppose there are requests to chunk i approximately every t time units. Let $q_s = 10$, $c = 3$, and the probability of failure in time interval t be $p = 1/10$. Then, the failure probability is on the order of 10^{-10}.

4.3 Defense Against Resource Exhaustion

Malicious nodes could flood $\mathcal{S}$ with bogus chunks of data, exhausting storage and denying service to legitimate users. *BitStore* must defend against this attack by allowing any node in $\mathcal{S}$ to verify whether the storage request is legitimate.

A legitimate client that wants to store a chunk on $\mathcal{S}$ constructs a message $M =< chunk\_hash, i >$, where $chunk\_hash$ is computed as in Section 4.2 and i is the index of the replica ($0 \leq i \leq q_s$), and presents it to its tracker. The tracker now generates the Chord key k using the hash function defined in Section 4.2 and returns $M' =< M, k >$, signed with its private key, to the client. The client presents M' and the tracker's signature with the request to store the chunk to the node in $\mathcal{S}$ responsible for storing the key k.

When a node in $\mathcal{S}$ receives a request to store a chunk, it performs two checks. First, it verifies the signature presented with the request using *its* own tracker (as all trackers know each others' public keys). Second, the node verifies that it is indeed responsible for storing the key k (obtained from M') by performing a Chord lookup. The node accepts the storage request only if both tests succeed.

To support these operations, each tracker must have a private/public key pair and be able to perform two operations: 1) sign a message at the request of any client currently connected to it using its private key; and 2) verify the signature on any message presented to it by a client, where the signature may be from a tracker from a different organization.

4.4 Privacy and Anonymity

To preserve privacy, nodes that contribute storage to $\mathcal{S}$ should not be able to discover the origin of content that they store (*e.g.*, the `.torrent` file for the chunks of any file they are storing). *BitStore* solves this problem by encrypting all chunks stored in $\mathcal{S}$, for example by using the secret shared between all members of a given swarm. This encryption also prevents nodes from discriminating against serving different types of content, and also provides the node plausible deniability. (Other systems have also provided similar guarantees [12, 29].)

5. Incentives

This section describes the incentive system that *BitStore* provides to encourage nodes to contribute resources to $\mathcal{S}$.

5.1 Overview of Token Scheme

$\mathcal{S}$ cannot rely on the BitTorrent-like tit-for-tat chunk trading scheme between nodes to ensure fairness: The node that is blocked and is requesting a chunk from some node in $\mathcal{S}$ will likely not have any data that the node offering the chunk wants. The rewards offered to a node providing service must be in a form *different from content*. Accordingly, *BitStore* uses currency—in the form of *tokens*—that are used to pay for chunks obtained from $\mathcal{S}$. The token scheme has the following properties:

- *BitStore* compensates a node in $\mathcal{S}$ only *after* the successful retrieval of a stored chunk. Otherwise, a dishonest node may discard content it was paid to store.
- A storage node should receive compensation in proportion to the services it has rendered. For instance, a node serving more chunks from $\mathcal{S}$ should receive higher priority while downloading chunks it needs.
- Tokens have real monetary value; this property automatically addresses concerns of replication, hoarding, etc. It also allows leechers to place a real dollar value on using blocks from $\mathcal{S}$ to become "unblocked".

Each token has a fixed denomination, but its *value* is subject to market forces much like in a real market. Any node may reuse tokens in subsequent sessions, but the same token cannot be used twice by any node; well-studied digital cash systems (*e.g.*, [27, Ch.6], [30], [31]) can be used to implement tokens.

5.2 Using Tokens

Any node can obtain and use tokens in the following way.

Obtaining Tokens. Tokens can be obtained in two ways. First, any node can buy tokens using real money (*e.g.*, using a credit card or a bank account) at exchange portals (run by the same organizations that run trackers) at the offered conversion rate (similar to foreign currency exchange). Maintaining portals offers a source of income (*e.g.*, advertisements) for these organizations. Second, a node can *earn* tokens by participating in $\mathcal{S}$ and serving requests.

Using Tokens. A node that wants to download a chunk from $\mathcal{S}$ first issues an `mget()`. On finding a node that has the chunk, the node participates in a *second price auction* with other nodes that are requesting the same chunk from that seller (Section 5.3 describes the auction mechanism in detail). The requesting node (the "buyer") can choose any seller among the replicas storing the chunk that gives it the best price and performance, and the seller is free to deny service if the winning buyer does not meet its reserve price. At the conclusion of the auction, the winning bidder's tokens are exchanged for the chunk using a secure mechanism [7].

5.3 Auction Mechanism

A secure, sealed-bid auction is held by a seller at regular time intervals. Note that even losers in an auction can benefit since the winner is a peer in B's swarm: B can then obtain that chunk with conventional tit-for-tat block exchange.

Second-Price Auction and Reserve Price. The basic auction protocol is described in case 3 of Figure 2. This second-price auction has the desirable *truthfulness* property that it gives buyers the incentive to bid their true valuation of the chunk [26]. Each auction also has a *reserve price*, the minimum payment the seller demands for the chunk, that is known only to the seller. If the auction has only one buyer, then the block is sold at the reserve price, assuming the reserve is met.

If no bidder is able to meet the reserve price, the chunk is not sold. Buyers however are free to try again later (possibly after earning or buying tokens). Alternatively, sellers may altruistically agree to sell the chunk below its reserve price (or specifically, for free); this mechanism is equivalent to *optimistic unchokes* in BitTorrent. Because sellers are allowed to sell the same chunk twice (described next), altruism does not suffer disincentives.

Bidding for a Set of Chunks in Parallel Auctions. Because each node ultimately cares about downloading an entire file rather than individual chunks, a node will typically value a *set* of chunks from $\mathcal{S}$ higher than the sum of its valuations for individual chunks. To ensure that nodes bid their valuation for the collection of chunks, we propose following mechanism. Assume that a node needs k chunks of a file from $\mathcal{S}$ for a total valuation of V. The node can bid $\frac{V}{k}$ at each chunk's auction. If it loses some auctions, these auctions will be won by nodes in its own swarm. With high probability, the node will obtain these chunks through exchanges and thereby the entire set of k chunks it desired, for less than V.

Double-selling. Although a seller serves only one buyer (the winning bidder) at a time, it may serve the same chunk to multiple buyers over time. This is advantageous to *BitStore* as the seller receives payment for the *service*, not the good. Of course, the seller must still offer reasonable prices for this service, since the buyers may choose among all given sellers before buying any chunks; moreover, a seller may not be able to sell a chunk too frequently as if a chunk was sold recently, a copy is likely to be in the corresponding swarm.

6. Conclusion

This paper has presented the design of *BitStore*, a distributed storage system for solving the Blocked Leecher Problem. *BitStore* builds on previous work in DHT-based cooperative storage systems, but extends these systems to provide replica maintenance, robustness against resource exhaustion attacks, and strong privacy and anonymity guarantees.

BitStore leverages the fact that the total number of peers in a BitTorrent network is larger and more stable than nodes in any individual swarm to construct a storage system for blocks that would otherwise be unavailable when seeders depart. *BitStore* allows nodes in the BitTorrent network to contribute resources in exchange for tokens, which the node can later exchange either to unblock its own download (possibly for a different file) for money. *BitStore* requires no modifications to existing BitTorrent protocols and can thus be easily added as an auxiliary system to the already widely deployed base of BitTorrent networks.

REFERENCES

[1] Cachelogic Estimate for BitTorrent's Contribution to Internet Traffic. `http://in.tech.yahoo.com/041103/137/2ho4i.html`, 2005.

[2] Demonoid Torrent Tracker. `http://www.demonoid.com/`, 2007.

[3] Mininova Torrent Tracker. `http://www.mininova.org/`, 2007.

[4] Mininova Torrent Tracker Statistics. `http://www.mininova.org/stats`, 2007. Retrieved on 23 February 2007.

[5] The Pirate Bay. `http://thepiratebay.org/`, 2007.

[6] AIYER, A. S., ALVISI, L., CLEMENT, A., DAHLIN, M., MARTIN, J.-P., AND PORTH, C. Bar fault tolerance for cooperative services. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles* (New York, NY, USA, 2005), ACM Press, pp. 45–58.

[7] ASOKAN, N., SCHUNTER, M., AND WAIDNER, M. Optimistic protocols for fair exchange. In *ACM Conference on Computer and Communications Security* (1997), pp. 7–17.

[8] BHARAMBE, A. R., HERLEY, C., AND PADMANABHAN, V. N. Analyzing and Improving a BitTorrent Network's Performance Mechanisms. In *Proceedings of 25th IEEE International Conference on Computer Communications (INFOCOM 2006)* (April 2006), pp. 1–12.

[9] BLAKE, C., AND RODRIGUES, R. High availability, scalable storage, dynamic peer networks: Pick two. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)* (Lake George, NY, Oct. 2003).

[10] CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M., NANDI, A., ROWSTRON, A., AND SINGH, A. SplitStream: High-bandwidth content distribution in cooperative environments. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)* (Lake George, NY, Oct. 2003).

[11] CASTRO, M., AND LISKOV, B. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems 20*, 4 (Nov. 2002), 398–461.

[12] CLARKE, I., SANDBERT, O., WILEY, B., AND HONG, T. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. the Workshop on Design Issues in Anonymity and Unobservability* (Berkeley, CA, July 2000).

[13] COHEN, B. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems* (Berkeley, CA, USA, June 2003).

[14] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)* (Banff, Canada, Oct. 2001).

[15] The eMule Project. `www.emule-project.net/`.

[16] FELDMAN, M., LAI, K., STOICA, I., AND CHUANG, J. Robust incentive techniques for peer-to-peer networks. In *EC '04: Proceedings of the 5th ACM conference on Electronic commerce* (New York, NY, USA, 2004), ACM Press, pp. 102–111.

[17] HABIB, A., AND CHUANG, J. Incentive mechanism for peer-to-peer media streaming. In *Twelfth IEEE International Workshop on Quality of Service (IWQOS), 2004.*

[18] IZAL, M., URVOY-KELLER, G., BIERSACK, E., FELBER, P., HAMRA, A., AND GARCES-ERICE, L. Dissecting bittorrent: Five months in a torrent's lifetime. In *Proceedings of the 5th Passive and Active Measurement Workshop.* (April 2004).

[19] KazaA filesharing network. `http://www.kazaa.com/`.

[20] LAI, K., FELDMAN, M., STOICA, I., AND CHUANG, J. Incentives for cooperation in peer-to-peer networks. In *Proceedings of the 5th ACM conference on Electronic commerce (EC), 2004* (2003).

[21] M. SIRIVIANONS AND J. H. PARK AND X. YANG AND STANISLAW JARECKI. Dandelion: Cooperative Content Distribution With Robust Incentives. In *Proc. USENIX Annual Technical Conference* (Santa Clara, CA, June 2007).

[22] MALKHI, D., AND REITER, M. Byzantine quorum systems. In *Proceedings of the 29th annual ACM symposium on Theory of computing(STOC)* (1997), pp. 569–578.

[23] MICALI, S., AND RIVEST, R. L. Micropayments Revisited. *Lecture Notes in Computer Science*, 2271 (2002), 149–263.

[24] PIATEK, M., ISDAL, T., ANDERSON, T., KRISHNAMURTHY, A., AND VENKATARAMANI, A. Do Incentives Build robustness in BitTorrent? In *Proc. 4th USENIX NSDI* (Cambridge, MA, Apr. 2007).

[25] POUWELSE, J. A., GARBACKI, P., EPEMA, D. H. J., AND SIPS, H. J. The Bittorrent P2P file-sharing system: Measurements and analysis. In *4th Int'l Workshop on Peer-to-Peer Systems (IPTPS)* (Feb 2005).

[26] ROBERT GIBBONS. *Game Theory for Applied Economists*. Princeton University Press, 1992.

[27] SCHNEIER, B. *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition*. Wiley, October 1995.

[28] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM* (San Diego, CA, Aug. 2001).

[29] WALDMAN, M., AND MAZIÈRES, D. Tangler: A censorship-resistant publishing system based on document entanglements. In *Proc. 8th ACM Conference on Computer and Communications Security* (Philadelphia, PA, Nov. 2001).

[30] Digital Cash. `http://www2.sims.berkeley.edu/courses/is204/f97/GroupE/cash.html`, 1997.

[31] Micropayment Products. `http://www2.sims.berkeley.edu/courses/is204/f97/GroupE/micro.html`, 1997.

APPENDIX

A. Estimate of Disk Space Requirement

The statistics retrieved from the Mininova torrent aggregator [4] on March 21, 2007, show the following:

Total number of seeders, s: 3,964,445

Total number of leechers, l: 4,950,150

Total users, $u = l + s$: 8,914,595

Total torrent files, t: 3,583,654

Average file size, f: 60.15 MB

For a uniform replication of each file 10 times, the amount of disk space contributed by each user (assuming uniform distribution) = $(t \times f \times 10)/u) \approx$ 242MB.

B. Proof Sketch of Failure Probability

Theorem B.1 *Failure Probability in case of an* `mget` *request by a leecher for chunk i is proportional 2^{-q_i}.*

PROOF SKETCH. Assume that there has been an `mget` request for chunk i in the previous T units of time, where T is proportional to the average lifetime of nodes.

By our protocol, at the time of this `mget` request, at least c copies of chunk i were received an `mput` request. Moreover, these c where chosen uniformly at random from the q_i. Therefore, the probability that these c stored a chunk but none of the other $q_i - c$ did is $\frac{r!(n-r)!}{n!}$ where $r = c$ and $n = q_i$. For appropriate c, this is exponentially small in q_i; this is because, after every `mget`, c of these are selected at random, so over k such requests, each of the q_i nodes is checked $\frac{kc}{q_i}$ times in expectation (c chosen such that this ratio is constant bigger than 1); we assume that the probability of a random node failing within k such requests is small.

This tells us that if every `mget` request is accompanied by c `mput` requests we can guarantee that immediately after the `mget`, with high probability, a large fraction of the q_i replicas indeed store the chunk. Now, the probability that all of these nodes fail, before the next `mget` to chunk i (given that this request is soon enough), is p^t where p is the probability of failure of a single node and t is the number of replicas of the chunk. Using this, if the probability of a random node failing between two successive `mget` requests to chunk i is $\frac{1}{2}$ and t is a constant fraction of q_i, the result in the theorem follows. ∎