Dandelion: Cooperative Content Distribution with Robust Incentives

Michael Sirivianos Xiaowei Yang Stanislaw Jarecski
Department of Computer Science
University of California, Irvine
{msirivia,xwy,stasio}@ics.uci.edu

Abstract

Online content distribution has increasingly gained popularity among the entertainment industry and the consumers alike. A key challenge in online content distribution is a cost-efficient solution to handle demand peaks. To address this challenge, we propose Dandelion, a system for robust cooperative (peer-to-peer) content distribution. Dandelion explicitly addresses two crucial issues in cooperative content distribution. First, it provides robust incentives for clients who possess content to serve others. A client that honestly serves other clients is rewarded with credit that can be redeemed for future downloads at the content server. Second, Dandelion discourages unauthorized content distribution. A client that uploads to another client is rewarded for its service only after the server has verified the other client's legitimacy. Our preliminary evaluation of a prototype system running on commodity hardware with 1 Mbps uplink and 1 Mbps downlink indicates that Dandelion can achieve aggregate client download throughput three orders of magnitude higher than the one achieved by an HTTP/FTP-like server.

1 Introduction

Content distribution via the Internet is becoming increasingly popular among the industry and the consumers alike. A survey showed that Apple's iTunes music store sold more music than Tower Records and Borders in the US in the summer of 2005 [18]. A number of key content producers, (e.g. CBS, Disney, Universal), are now selling films online [2, 3, 6].

A challenging issue for online content distribution is a cost-effective solution to handle peak usage by promotions or new releases. A 45-minute DVD-quality episode easily exceeds one GB. Even if each user is provisioned with a 1 Mbps, it takes more than two hours to download 1 GB. Overprovisioning for one additional user during peak usage may require at least an additional 1Mbps bandwidth, which often costs up to $100 per month [5, 13]. However, a TV episode is commonly sold at less than two dollars. One solution is to purchase service from a content distribution network (CDN) such as Akamai. Yet, CDNs' services are costly too, and free CDNs such as Coral, CoDeen, and CobWeb [11, 20, 26, 31] lack a viable economic model to scale.

This work explores a cost-effective approach for handling flashcrowds. We present the design and a preliminary evaluation of Dandelion, a cooperative content distribution system. Rather than using a third party service, a Dandelion server utilizes its clients' bandwidth. During a flash crowd event, a server redirects a request from a client to the clients that have already downloaded the same content. This approach is similar in spirit to previous work on cooperative content distribution [12, 16, 24, 27, 28], most notably BitTorrent [8]. However, with the exception of BitTorrent, the above approaches do not provide incentives for a client to upload to other clients. BitTorrent employs rate-based tit-for-tat incentives, but these are susceptible to manipulation [15, 17, 25] and do not motivate clients to upload content after the completion of their download (i.e., seeding).

The primary contribution of our work is that we provide robust incentives for clients to upload to others. By robust, we mean that the incentive mechanism does not rely on clients being altruistic or honest. Its secondary contribution is that Dandelion discourages unauthorized content sharing. Our design gives no incentives to clients to upload to unauthorized clients, but provides explicit rewards for them to upload to authorized clients, e.g., clients that have purchased content at a server.

Dandelion's incentive mechanism is based on a cryptographic fair exchange mechanism, which uses only efficient symmetric cryptography. A client uploads content to other clients in exchange of virtual credit. The credit can be redeemed for future service by other clients, or for service by the server itself, or other rewards. This incentive mechanism discourages unauthorized content exchange, because a client is rewarded for its service only after the server has verified that the client has uploaded to an authorized client.

We have implemented a prototype of a Dandelion client and server and conducted a preliminary evaluation on PlanetLab [7]. We compare the throughput of a Dandelion server with a server that runs a simple request-response protocol, such as HTTP. Our preliminary evaluation shows that Dandelion can improve the throughput of a commodity PC server with 1 Mbps uplink and 1 Mbps downlink bandwidth by three orders of magnitude. However, as a trade-off for providing robust incentives and discouraging unauthorized content distribution, a Dandelion server is less efficient than a BitTorrent tracker. As a result, a Dandelion system is less scalable than BitTorrent, with respect to the number of active clients supported by a single server/tracker.

The rest of this paper is organized as follows. Section 2 describes the design of Dandelion. Section 3 briefly discusses our implementation and its performance. Section 4 compares our work with related work. We conclude in Section 5. In the Appendix we provide a detailed description of our protocol and discuss its security.

2 Design

This section describes the design of Dandelion at a high-level. In Appendix, we describe the protocol in more detail.

2.1 Overview

The premise of our design is that a low server may have limited outgoing bandwidth but sufficient CPU power, and memory to execute many short cryptographic operations and maintain TCP connection state with its clients under a flash crowd event.

A Dandelion server can be used to distribute both small and large static files, depending on the specifics of its deployment (see Section 2.3). It behaves similar to a web/ftp server under normal work load, responding to clients' requests with content. When a Dandelion server is overloaded, it enters a *peer-serving* mode. Upon receiving a request, the server redirects the client to clients that are able to serve the request.

A Dandelion server maintains a virtual economy. It rewards cooperative clients that upload to others with virtual credit to provide robust incentives. The credit is used as "virtual money" to purchase future downloads from other clients or from the server itself (at a high credit cost when the server is overloaded), or used as other types of rewards.

Similar to BitTorrent, a Dandelion server splits a large file into multiple chunks, and disseminates them independently. This allows clients to participate in uploading chunks as soon as they receive a small portion of the file, increasing the efficiency of the distribution pipeline. Furthermore, this incentivizes clients to upload chunks to others, as they need credit to acquire the missing ones.

2.2 Robust incentives

A key challenge in designing a credit system is to prevent client cheating, while keeping both a server and a client's processing and bandwidth costs low. A dishonest client

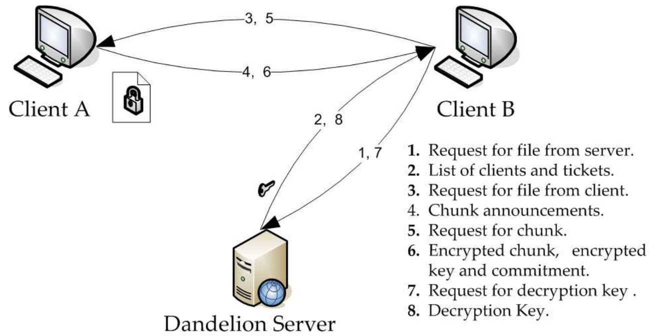

Figure 1: The peer-serving protocol. The numbers on the arrows correspond to the listed protocol messages. The messages are sent in the order they are numbered.

that does not upload to others or uploads garbage may attempt to claim credit at the server, and to be robust, the server must not award credit to such cheating behavior. To address this challenge, Dandelion employs a cryptographic fair exchange mechanism. A Dandelion server serves as the trusted third party mediating the exchanges of content for credit among its clients. When a client *A* uploads to a client *B*, it sends encrypted content to client *B*. To decrypt, *B* must request keys from the server. The requests for keys serve as the "proof" that *A* has uploaded some content to *B*. Thus, when the server receives a key request, it credits *A* for uploading to *B*, and charges *B* for the content.

A problem occurs if a malicious client *A* sends invalid content to *B*. *B* can discover that the content is invalid only after receiving the decryption key and being charged. To address this problem, our design includes a non-repudiable complaint mechanism. If *A* intentionally sents garbage to *B*, *A* cannot deny it. In addition, *B* is prevented from falsely claiming that *A* has sent it garbage. For clarity, we describe the complaint mechanism after we describe the normal message exchange in a Dandelion system.

Figure 1 shows how messages are exchanged in a Dandelion system. We assume that each client has a password-protected account with the server and that it establishes a secure channel (e.g. SSL), over which it obtains shared session keys with the server. During a flash crowd event, the Dandelion server keeps track of the clients that are currently downloading or seeding offered files. The message exchange proceeds as follows:

Step 1: A client (*B* in Figure 1) sends a request for a file to the server.

Step 2: When the server receives the request, it returns digests of the file chunks for integrity checking [8], a random list of other clients that can serve the file, and cryptographic authorizations, namely tickets that enable *B* to request chunks from these clients.

Step 3: Upon receiving the server's response, *B* connects to the listed clients to request the file. We use client A as an example in Figure 1.

Step 4: If B's tickets verify that the server has authorized B to request chunks from A, B and A will run a chunk selection protocol similar to that in BitTorrent [8]. A reports periodically to B what chunks it has. B determines which chunks it wishes to download and from which peers according to a chunk scheduling algorithm such as *rarest first*.

Step 5: B sends a request for the chunk to A.

Step 6: If B's ticket verifies, A chooses a random key k, and encrypts it with the session key K_{SA}, it shares with the server. Client A sends to B the chunk encrypted with k, the encryption of the key k, and its cryptographic commitment to the encrypted chunk. A generates the commitment by computing a message authentication code (MAC), keyed with the shared session key K_{SA}, over the digest of the encrypted chunk and the encryption of k

Step 7: To retrieve k, B sends a decryption key request to the server. The request contains the encryption of the key k, a digest of the encrypted chunk, and A's commitment.

Step 8: Upon receiving B's request, the server checks whether A's commitment matches the one computed over B's digest of the encrypted chunk and the encryption of key k, using K_{SA}. If the commitment verifies and B has sufficient credit, the server sends the key to B. At the same time, it rewards A with credit and charges B.

If A's commitment does not verify, the server cannot determine whether the discrepancy is caused by a transmission error, or client A or B is misbehaving. The server simply warns B of the discrepancy, and does not return the encryption key k. It updates neither A's or B's credit. B can re-request the chunk from A or try another client.

If B repeatedly receives invalid commitments from A, it should disconnect from A and blacklist it. Similarly, if the server repeatedly receives decryption requests from B with invalid commitments from a specific A, the server knows that B is misbehaving because B should have blacklisted A. The server will blacklist B.

Next, we explain the complaint mechanism. After B receives the key k, it decrypts the chunk and validates its integrity. If the chunk is invalid, B can complain to the server, and A cannot repudiate it. This is because B's complaint message contains A's commitment, the digest of the encrypted chunk, and the encryption of key k, all received in the message from A in Step 6. The server can easily validate whether A has sent the commitment, as the commitment is a MAC computed with the session key K_{SA} shared between the server and A. B cannot forge a valid commitment. If the commitment fails, the server knows that B is misbehaving, since it should have abandoned the transaction in step 8. If the commitment verifies, A cannot repudiate that it has sent the commitment to B. All the server needs to check is whether A has computed the commitment over a valid chunk. To verify this, the server retrieves and encrypts the chunk that B complains about, using the key k and computes the MAC using the shared key K_{SA}. If this recomputed commitment matches A's commitment, it proves that A has sent the valid content, and B is framing A; otherwise, it proves that A has sent invalid content to B. A misbehaving client is blacklisted by the server and its peers. Requests involving the misbehaving client are no longer processed. Future complains concerning the misbehaver are ruled against it.

2.3 Credit Management

Dandelion can be used for both free and paid content. In both cases, clients spend $\Delta_c > 0$ credit units for each chunk they download from a client and earn $\Delta_r > 0$ credit units for each chunk they upload to a client. A client can acquire a file chunk only if its credit is greater or equal to the chunk's cost. To prevent collusions we set $\Delta_c = \Delta_r$, so that two colluders cannot increase the sum of their credit.

In the free-content case, each client is given an initial small amount of credit when it first registers with the server. This initial credit enables a new user to download only a few chunks when it joins the Dandelion swarm. Thus, a client is incentivized to upload to others in order to accumulate credit to be used towards downloading the complete content. Consequently, clients upload chunks proportionally to the number of chunks they download.

In the paid content case, a provider may redeem a client's credit for monetary rewards, such as discounts on content prices or service membership fees, similar to the mileage programs of airline companies. Therefore, a client could be awarded sufficient initial credit to download the complete file from other clients, and it would still be motivated to earn credit.

2.4 Discouraging unauthorized content distribution

In Dandelion, rational authorized clients are discouraged from serving content to unauthorized clients. This is because a server does not award them credit for illegitimate transactions. Clients are able to verify the legitimacy of requests for service (as described in Section 2.2), hence they can avoid wasting bandwidth to send encrypted chunks to unauthorized peers. Furthermore, due to this ability, clients can be held liable if they choose to send plaintext contents to unauthorized clients. These properties discourage users from using Dandelion for illegal content replication and make our solution appealing to distributors of copyright-protected digital goods.

3 Preliminary Evaluation

We implemented a prototype of Dandelion in C under Linux, and conducted a preliminary evaluation of its per-

Dandelion Server

Dandelion Operation	Size	Time (ms)
Verify decryption key request ticket (MAC)	125 bytes	0.018
Decrypt decryption key	40 bytes	0.087
Transmit decryption key response	92 bytes	∼ 1.36
Receive decryption key request	148 bytes	∼ 1.81
Query and update credit base (SQLite)	N/A	1.08
Receive chunk request	56 bytes	∼ 1.07
Transmit chunk	256 KB	∼ 2209

Dandelion Client

Dandelion Operation	Size	Time (ms)
Encrypt/decrypt chunk	256 KB	4.1
Encrypt/decrypt chunk	16 KB	0.35
Commit to encrypted chunk (hash and MAC)	256 KB	1.45

Table 1: Timings of per-chunk Dandelion operations. The server and client is rate-limited to emulate Ethernet II 1 Mbps uplink and 1 Mbps downlink.

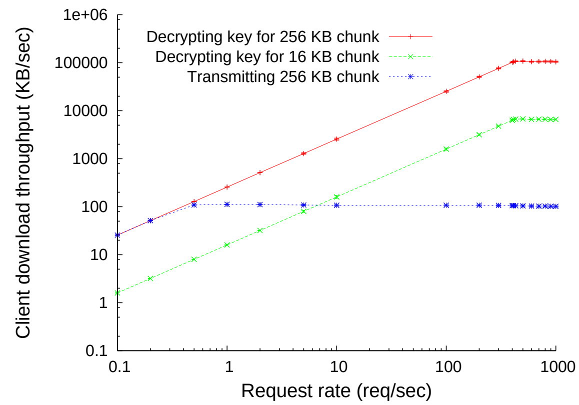

Figure 2: The *y* axis shows the achievable aggregate download throughput of Dandelion clients when the server responds to: a) requests for keys used to decrypt 256 KB encrypted chunks; b) requests for keys used to decrypt 16 KB encrypted chunks; and c) requests for chunks. The *x* axis is the specified aggregate client request rate.

formance on PlanetLab. This section describes our implementation and the results of our PlanetLab experiments.

3.1 Prototype Implementation

We implemented Dandelion's cryptographic operations using the *openssl* C library and the credit management system using the lightweight database engine of the *sqlite* library.

Our server implementation draws from the Flash [19] web server's Asymmetric Single Process Event Driven Architecture and the Staged Event Driven Architecture [10]. Both architectures assign thread pools to specific tasks.

When a disk read or a database operation is required by a request, Dandelion's main thread reads requests from the network and dispatches them to a synchronized producer-consumer queue served by a pool of *disk access* or *database access* helper threads, respectively. When a helper thread finishes its operations, it dispatches the request to another thread pool (next stage) for subsequent processing. We use the zero-copy *sendfile()* system call, which is called by the *disk access* threads. The network operations use TCP, are asynchronous and are are executed by the thread responsible for the last stage of request processing.

This design exploits parallelism and maintains good performance when both small and cached files or large disk-residing files are requested from the server itself. In addition, it does not bind the number of concurrent connections or pending requests to the number of processes/threads that the OS can efficiently accomodate simultaneously.

3.2 Experimental Results

We first evaluate the computational costs of a Dandelion server. In a flash crowd event, the main task of a Dandelion server is to process key decryption requests and send short responses to those requests. To process one decryption request, a server performs one HMAC operation and one block cipher decryption on small messages. Furthermore, it performs one query and two update operations on a credit database. Lastly, it transmits the decryption key.

In our experiments, we deployed a Dandelion system with one server and 100 clients. The server runs on Linux 2.6.14 on a 1.7 GHZ/2 MB Pentium M CPU and 1 GB RAM. To stress our design and emulate a typical resource-limited server with Ethernet II 1 Mbps uplink and 1 Mbps downlink, we rate-limited our server's and client's upload and download rate at the application layer.

We let the Dandelion clients send the following two types of requests to the server and benchmarked the client download throughput along with the processing costs. The first type was requests for decryption keys, which emulate the load on the server during peer-serving. The second type was requests for file chunks directly from the server. The file resided in memory in its entirety for the duration of the experiment. Each client sent requests at a rate ranging from 0.001 to 10 requests/sec. As the request rate increased, the client would send a new request prior to receiving the complete response from the server. Also as the request rate increases and the server's receiver buffers get full, clients would not send new requests at the rate specified because the server would advertizes 0 byte window or the previous request would be pending tobe sent. For each rate, the experiment duration was 10 minutes and the results were averaged over 10 experiments.

Table 1 shows the cost for each operation. As can be seen, the cryptographic operations of Dandelion are highly efficient, as only symmetric cryptography is involved. From these results we conclude that in our experimental configuration the server's bottleneck is most likely to be its download link. A Dandelion client can encrypt and decrypt a 256 KB chunk much faster than download it or transmit it at 1 Mbps.This result suggests that the client's processing overhead does not affect its upload or download throughput.

Figure 2 compares the case in which Dandelion clients send decryption key requests to a server, as if they peer-serve each other, with the case in which clients request the file directly from the server, i.e., the HTTP/FTP-like downloading. The curves show that a Dandelion server running on a commodity PC with a 1 Mbps Ethernet uplink and 1 Mbps downlink can process up to $\sim$420 decryption key requests per second, effectively serving up to $\sim$1680 clients that download 256 KB chunks at 64 KB/s from other clients. They also show that with a chunk size of 256 KB, the Dandelion clients' download throughput would be almost 990 times higher than the throughput yielded when the clients request the file directly from the server. A smaller chunk size reduces the performance gain, as a server must process more decryption key requests.

The cost of a complaint is higher because it involves reading a chunk, encrypting it with the sender client's key and hashing the encrypted chunk. However, the server blacklists misbehavers, thus it does not repeatedly incur the cost of complaints sent by them.

4 Related Work

This section briefly compares our work with related work.

Swarm file downloading protocols. Dandelion is inspired by swarm downloading protocols such as Bittorrent [8] and Slurpie [24]. A key difference of our work is its robust incentive mechanism. Slurpie does not provide incentives for peer-serving. Although Bittorrent employs rate-based "tit-for-tat" incentives, these do not punish free riders [15] due to the specifics of its unchoking mechanism. In addition, a free rider can enhance its advantage by obtaining a larger than normal initial partial view of the BitTorrent network. In this way, a peer can discover many seeders and choose to connect to them only [17], increasing his download rate. It can also increase the frequency with which it gets optimistically unchoked by connecting to all leechers in its large view [25].

Furthermore, as there is no robust mechanism to motivate seeding in BitTorrent, the number of clients that seed for long periods of times is very small [21]. In contrast, credit in a Dandelion system provides robust incentives for clients to seed files, which will improve file availability and download completion times.

Lastly, Dandelion has the desirable feature that rational clients have no incentives to serve unauthorized peers, as in such case the server will not reward them. In BitTorrent, content access policies are enforced by requiring password-based authentication with the tracker. However, an unauthorized peer can join the network simply by finding a single colluding peer that is willing to share its swarm view with it. The unauthorized peers can then download content from authorized peers, which have the incentives to serve them as long as the unauthorized peer is tit-for-tat compliant. As a result, a single authorized but misbehaving peer can facilitate illegal content replication at a large scale. In an upcoming BitTorrent version, access policies are implemented by accelerating legitimate content transfers through the use of strategically placed caches, which can be accessed only by authorized clients [1, 4]. Our scheme does not require third party infrastructure.

Escrow services in peer to peer networks. Horne et al. [14] proposed an encryption-based fair-exchange scheme for peer-to-peer file exchanges. Dandelion shares similarities regarding motivation and the general approach with their work, but differs in specific protocol design. Their scheme divides and transmits a file in chunks to enable erasure-code-based techniques for detecting cheaters that upload invalid content, whereas we divide files to support efficient and incentivized peer-to-peer distribution. Their scheme detects cheating with probabilistic guarantees, whereas Dandelion deterministically detects and punishes cheaters. In addition, their scheme requires that all chunks for a given file come from a single peer, which renders the distribution pipeline inefficient.

Fair-exchange schemes. Among the proposed solutions for the classic cryptographic fair-exchange problem, our scheme bears the most similarity with the one by Zhou et al. [32]. Their scheme also encrypts the content to be exchanged and uses an online trusted third party (TTP) to relay the decryption key. A key difference is that Zhou et al.'s scheme uses public key cryptography for encryption and for committing to messages, and both of the exchange parties need to communicate with the TTP for each transaction. In contrast, our scheme uses efficient symmetric key encryption, and only one client needs to communicate with the TTP per transaction. The technique they use to determine whether a message originates from a party is similar to the one used by our complaint mechanism, but our work also addresses the specifics of determining the validity of the message.

Pairwise credit-based incentives. Swift [29] introduces a pairwise credit-based trading mechanism (barter) for peer-to-peer file sharing networks and examines the available peer strategies. Scrivener [9] is also an architecture in which peers maintain pairwise credit balances to regulate content exchanges among each other. In contrast, a Dandelion server maintains a central credit bank for all clients.

Global credit-based incentives. Similar to Dandelion, Karma [30] employs a global credit bank, with which clients maintain accounts. It distributes the credit auditor set of a peer among the peer's k closest neighbors in a DHT overlay [22]. Karma uses certified-mail-based [23] fair exchange of content for reception proofs, which requires both peers to communicate with the mediating auditor set for each exchange. Unlike Dandelion, Karma requires public key cryptographic operations at the peer side. Karma provides probabilistic guarantees with respect to the integrity of the credit-base. In the presence of numerous malicious bank nodes or in a highly dynamic network, the credit-base becomes difficult to maintain reliably.

5 Conclusion and Future Work

This paper describes a cooperative content distribution system: Dandelion. Dandelion's primary function is to offload a server during a flash crowd event, effectively increasing availability without overprovisioning. A server delegates a client with available resources to serve other clients. We use a cryptographic fair-exchange technique to provide robust incentives for client cooperation. The server rewards a client that honestly serves other clients with credit. A client can redeem its credit for further service or monetary rewards. In addition, the design of Dandelion discourages unauthorized content exchange. Since a server mediates all fair exchanges, clients who serve unauthorized requests are not rewarded, therefore it is in their best interests not to waste their upload bandwidth to serve unauthorized clients. A preliminary evaluation shows that Dandelion has low processing and bandwidth costs on the server side. A resource-limited server may support a few thousand simultaneous clients. We are currently fine-tuning Dandelion's prototype implementation on PlanetLab.

6 Acknowledgements

We thank Nikitas Liogkas and the anonymous reviewers for their useful feedback on early drafts of this paper. We also thank Jonathan Park and Rex Chen for their assistance with PlanetLab experiments.

References

[1] Bittorrent announces authorized rollout. `http://www.slyck.com/news.php?story=1090`, Feb. 2006.

[2] CBS to sell new survivor episodes on own site. `http://www.msnbc.msn.com/id/11134875/`, Feb. 2006.

[3] Disney does big business selling tv episodes online. `http://www.showbizdata.com/contacts/picknews.cfm/40484/DISNEY_DOES_BIG_%BUSINESS_SELLING_TV_EPISODES_ONLINE`, Jan. 2006.

[4] Ntl, bittorrent and cachelogic announce joint technology trial. `http://www.cachelogic.com/news/pr100206.php`, 2006.

[5] Quote from PACIFIC BELL: $18000 per month for an OC3 line. `http://shopforoc3.com/`, Mar. 2006.

[6] Universal announces a new download-to-own service. `http://edition.cnn.com/2006/TECH/03/23/movie.download/index.html`, Mar. 2006.

[7] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. In *SIGCOMM CCR*, pages 3–12, 2003.

[8] B. Cohen. Incentives build robustness in bittorrent. In *P2P Econ*, 2003.

[9] P. Druschel, A. Nandi, T.-W. J. Ngan, A. Singh, and D. Wallach. Scrivener: Providing incentives in cooperative content distribution systems. In *Middleware*, 2005.

[10] M. W. et al. Seda: An architecture for well-conditioned, scalable internet services. In SOSP, 2001.

[11] M. J. Freedman, E. Freudenthal, and D. Mazires. Democratizing content publication with coral. In *NSDI*, March 2004.

[12] C. Gkantsidis and P. Rodriguez. Network coding for large scale content distribution. In *INFOCOM*, 2005.

[13] J. Gray. Distributed computing economics. Technical report, Microsoft Research, 2003. MSR-TR-2003-24.

[14] B. Horne, B. Pinkas, and T. Sander. Escrow services and incentives in peer-to-peer networks. In *EC*, pages 85–94, 2001.

[15] S. Jun and M. Ahamad. Incentives in bittorrent induce free riding. In *P2P Econ*, pages 116–121, 2005.

[16] K. Kong and D. Ghosal. Pseudo-serving: a user-responsible paradigm for internet access. In *WWW*, pages 1053–1064, 1997.

[17] N. Liogkas, R. Nelson, E. Kohler, and L. Zhang. Exploiting bittorrent for fun (but not profi t). In *IPTPS*, 2006.

[18] S. Morris. iTunes outsells CD stores as digital revolution gathers pace. `http://arts.guardian.co.uk/netmusic/story/0,13368,1649421,00.html`, Nov. 2005.

[19] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An effi cient and portable Web server. In *USENIX*, 1999.

[20] K. Park and V. S. Pai. Scale and performance in the coblitz large-fi le distribution service. In *NSDI*, 2006.

[21] J. A. Pouwelse, P. Garbacki, D. H. J. Epema, and H. J. Sips. The bittorrent p2p fi le-sharing system: Measurements and analysis. In *IPTPS*, pages 205–216, 2005.

[22] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, 2001.

[23] B. Schneier. Applied Cryptography, 2nd edition, 1995.

[24] R. Sherwood, R. Braud, and B. Bhattacharjee. Slurpie: A cooperative bulk data transfer protocol. In *INFOCOM*, 2004.

[25] J. Shneidman, D. Parkes, and L. Massoulie. Faithfulness in internet algorithms. In *PINS*, 2004.

[26] Y. J. Song, V. Ramasubramanian, and E. G. Sirer. Cobweb: a proactive analysis-driven approach to content distribution. In *SOSP*, pages 1–3, 2005.

[27] T. Stading, P. Maniatis, and M. Baker. Peer-to-peer caching schemes to address fa sh crowds. In *IPTPS*, 2002.

[28] A. Stavrou, D. Rubenstein, and S. Sahu. A lightweight, robust p2p system to handle fa sh crowds. In *ICNP*, pages 226–235, 2002.

[29] K. Tamilmani, V. Pai, and A. Mohr. Swift: A system with incentives for trading. In *P2P Econ*, 2004.

[30] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer. Karma: A secure economic framework for p2p resource sharing. In *P2P Econ*, 2003.

[31] L. Wang, K. Park, R. Pang, V. S. Pai, and L. L. Peterson. Reliability and security in the codeen content distribution network. In *USENIX*, pages 171–184, 2004.

[32] J. Zhou and D. Gollmann. A fair non-repudiation protocol. In *IEEE Symposium on Research in Security and Privacy*, pages 55–61, 1996.

A Appendix

A.1 Detailed Protocol Description

This section provides a detailed description of the Dandelion peer-serving protocol.

A.1.1 Setting and Assumptions

We assume that the server S keeps a table matching any file F with a pool of available clients currently downloading or seeding the file. A client A gets a temporaty shared key K_{SA} with S. K_{SA} can be efficiently computed as $K_{SA} = H(K_S, \langle A\rangle, \langle i\rangle)$. The notation $\langle X\rangle$ denotes a client X's Dandelion ID, K_S is S's master secret key, H is a cryptographic hash function such as SHA-1, and $\langle i\rangle$ refers to a time period. Our protocol enables S to tolerate some lag in the $\langle i\rangle$ assumed by a client. The temporary shared keys are delivered from the server to the client over a secure channel. For every client, the server S maintains database entries of that client's credit, virtual money which can be used to purchase more services.

A.1.2 Client-Serving Protocol

The protocol starts with the client B sending a request for file $\langle F\rangle$ to S.

1) $B \longrightarrow S$: [server file request] $\langle F\rangle$

If B has access to F, S chooses a random short list of clients $\langle A\rangle_{\text{list}}$, which are currently downloading or seeding the file. Each list entry, besides the Dandelion ID of the client, also contains the clients inbound *internet address*. Also for every client in $\langle A\rangle_{\text{list}}$, S sends a ticket $T_{SA} = MAC_{K_{SA}}[\langle A\rangle, \langle B\rangle, \langle F\rangle, \text{ts}]$ to B. *MAC* is a message authentication code (e.g. an HMAC), ts is a timestamp and $\langle A\rangle$ is a client in $\langle A\rangle_{\text{list}}$. The tickets T_{SA} are only valid for a certain amount of time T (considering clock skew between A,S) and allow B to request chunks of file F from client's A. When T_{SA} expires and B still wishes to download from A it requests a new T_{SA} from S. As commonly done to maintain file integrity, S also sends the *SHA-1* hash $h_{\langle ch\rangle} = H(ch)$ for all chunks ch of the file F S may charge B for the issuance of tickets T_{SA} to prevent misbehaving clients from wasting server resources.

2) $S \longrightarrow B$: [server file response] $T_{SA\ \text{list}}, \langle A\rangle_{\text{list}}, h_{\langle ch\rangle\text{list}}, \langle F\rangle, \text{ts}, \langle i\rangle_S$

The client B forwards this request to each $A \in \langle A\rangle_{\text{list}}$

3) $B \longrightarrow A$: [client file request] $T_{SA}, \langle F\rangle, \text{ts}, \langle i\rangle_S$

If $current - time \le ts + T$ and T_{SA} is not in A's cache, A verifies if $T_{SA} = MAC_{K_{SA}}[\langle A\rangle, \langle B\rangle, \langle F\rangle, \text{ts}]$.[1] As long as B remains connected to A, it periodically renews its T_{SA} tickets. If the verification fails, A drops this request. Also, if $\langle i\rangle_S$ is greater than A's current epoch $\langle i\rangle_A$, A learns that it should renew its key with S soon. Otherwise, A caches T_{SA} and it starts running a protocol with B for file chunk selection. A reports periodically to B what chunks it has for as long as the timestamp is fresh. Also, B reports its available chunks to A and A can request them from B, after he retrieves T_{SB} from S. B determines which chunks it wishes to download and from which clients according to a chunk selection algorithm. For presentation purposes, each of the following messages involves one chunk, whereas in practice information for multiple chunks may be bundled in a message.

4) $B \longrightarrow A$: [client chunk request] $T_{SA}, \langle F\rangle, \langle ch\rangle, \text{ts}, \langle i\rangle_S$

B's requests are served as long as the timestamp is fresh and T_{SA} is cached or verifies. For each requested chunk, A retrieves and encrypts it using a symmetric-key encryption Enc, as $C = Enc^{\text{iv}_{\langle \text{ch}\rangle}}_{k_{\langle ch\rangle}}(ch)$, where $k_{\langle ch\rangle}$ is a randomly chosen key distinct for each chunk, and $\text{iv}_{\langle ch\rangle}$ is the encryption Initial Vector (IV). A encrypts the random key with the one it shares with the server, as $e = Enc^{\text{iv}_{\text{SA}}}_{K_{SA}}(k_{\langle ch\rangle}, \text{iv}_{\langle ch\rangle})$. Finally, A hashes the ciphertext C as $hc = H(C)$ and computes a MAC value $T_{AS} = MAC_{K_{SA}}[\langle A\rangle, \langle B\rangle, \langle F\rangle, \langle ch\rangle, e, hc, \text{ts}]$. Note that A can pre-compute several values $(k_{\langle ch\rangle}, e, C, hc)$, so the on-line cost of A can be reduced to one MAC computation.

5) $A \longrightarrow B$: [client chunk response]$T_{AS}, \langle F\rangle, \langle ch\rangle, e, C, \text{ts}, \langle i\rangle_A$

B retrieves C, computes its own hash $hc' = H(C)$ and forwards the following to S.

6) $B \longrightarrow S$: [decryption key request]$\langle A\rangle, \langle F\rangle, \langle ch\rangle, e, hc', \text{ts}, T_{AS}, \langle i\rangle_A$

If timestamp ts is fresh enough, ticket T_{AS} is not in S's cache, and $\langle i\rangle_A$ is not too much off, S checks if $T_{AS} = MAC_{K_{SA}}[\langle A\rangle, \langle B\rangle, \langle F\rangle, \langle ch\rangle, e, hc', \text{ts}]$, where key K_{SA} is computed using K_S, $\langle A\rangle$, and $\langle i\rangle_A$. The verification may fail either because hc' is invalid due to transmission error in step (5) or because either A or B are misbehaving. Since S is unable to determine which one is the case, it does not punish either clients. Yet it notifies B, which is expected to remove A from its client list in case A repeatedly sents invalid messages. If B keeps sending invalid decryption key requests, S penalizes him. If the verification succeeds, S caches T_{AS}, and checks whether B has sufficient credit. It also checks again whether B has access to the file F. If B is approved, it charges B and reward A. S also decrypts $(k'_{\langle ch\rangle}, \text{iv}'_{\langle ch\rangle}) = Dec_{K_{SA}}(e)$, and sends them to B.

7) $S \longrightarrow B$: [decryption key response]$\langle A\rangle, \langle F\rangle, \langle ch\rangle, (k'_{\langle ch\rangle}, \text{iv}'_{\langle ch\rangle}$

[1]The purpose of this check is to provide a simple mechanism for protecting A from DoS attacks from unauthorized clients and to allow clients to filter request for unauthorized file uploading.

B uses $(k'_{\langle ch\rangle}, \mathrm{iv}'_{\langle ch\rangle})$ to decrypt the chunk as $ch' = Dec_{k'_{\langle ch\rangle}}^{\mathrm{iv}'_{\langle ch\rangle}}(C)$. If the decryption fails or if $H(ch') \neq h_{\langle ch\rangle}$ (see item (2)), then B complains to S by sending the following message.

8) $B \longrightarrow S$: $[\text{complaint}], \langle A\rangle, \langle F\rangle, \langle ch\rangle, T_{AS}, e, hc', \text{ts}, \langle i\rangle_A$

S ignores this message if timestamp ts is not fresh enough (using much more liberal time interval then before) or if this complaint is already cached. If $T_{AS} \neq MAC_{K_{SA}}[\langle A\rangle, \langle B\rangle, \langle F\rangle, \langle ch\rangle, e, hc', \text{ts}]$ S punishes B, since B had already been notified in step (6) that T_{AS} is invalid. If T_{AS} verifies, S caches this complaint, recomputes K_{SA} as before, decrypts $(k'_{\langle ch\rangle}, \mathrm{iv}'_{\langle ch\rangle}) = Dec_{K_{SA}}(e)$ once again, retrieves ch from its storage, and encrypts ch himself using the above key and IV vector, $C' = Enc_{k'_{\langle ch\rangle}}^{\mathrm{iv}'_{\langle ch\rangle}}(ch)$. If the hash of the ciphertext $H(C')$ is equal to the value hc' that B sent to S, then S decides that A has acted correctly, B's complaint is unjustified, S drops this complaint request and blacklists B or charges B a large amount. Otherwise, S decides that B was cheated by A, removes A from its pool of active clients, blacklists or charges it, and issues an update that cancels the corresponding update on A's and B's credit. Lastly, B requests the chunk from another client or S itself.

A.2 Security Analysis

We claim the following security properties of our protocol:

1. If an honest client B gets charged (his credit decreases) by S, then B must have received correct chunk ch, even if the transaction involved a malicious client A. This is because B gets charged only if the data S gets in steps (6) and (8) verifies and if $hc' = H(C')$. Since hc' is a hash that B computes itself on C received from A, $C = C'$. Furthermore, since the same k, iv pair is used by S to encrypt ch into C' and by B to decrypt C into ch', then $C = C'$ implies that $ch' = ch$.

2. If an honest client A always encrypts chunk ch anew when servicing a request, then even if client B is malicious, if B gets ch in this protocol instance then A also gets credit from S. This is because if A encrypts ch using one-time key $k_{\langle ch\rangle}, \mathrm{iv}_{\langle ch\rangle}$, then B sees $k_{\langle ch\rangle}, \mathrm{iv}_{\langle ch\rangle}$ only in the encrypted form e. The only way for B to get it (short of stealing key K_{SA} from A or S) is to get it decrypted by S, in which case S will log a charge against B. The only way B can possibly avoid this charge is by sending (8) which includes T_{AS} and hc' s.t. $T_{AS} = MAC_{K_{SA}}[\langle A\rangle, \langle B\rangle, \langle ch\rangle, e, h', \text{ts}]$, but such that $hc' \neq H(C')$ where C' is computed by S in that step. However, since we consider this attack only against an honest A, the T_{AS} MAC value will verify only if all the values it includes are the ones that A sent to B, and if hash hc' is correctly computed on ciphertext C included in that transfer. But if that's the case then S will decrypt e to the same $k_{\langle ch\rangle}, \mathrm{iv}_{\langle ch\rangle}$ pair that A used, hence S's encryption C' will be the same as the C that A computed. Consequently, hc' will be equal to $H(C')$, hence B is not able to reverse its charge.

3. If A pre-computes only one encryption of some chunk ch and services requests for that file always using the same ciphertext (C, e), then A runs some risk that colluding B's can attempt to use A to download ch with only one of the B's charged for it. Namely, the colluding clients B's have some chance of getting tickets to the same client A from S, so each of them would receive the same encryption C of ch from A. Then one B can incur a charge to retrieve key $k_{\langle ch\rangle}$, but it can share this key with the remaining colluders. The chance of success in such attack decreases if the list of the clients returned by S is short and if A pre-computes many ciphertext tuples $(k_{\langle ch\rangle}, e, C, h)$ for the same ch, and services a request by choosing one of them at random. Note that A can individually adjust how much to pre-compute, or even to always encrypt ch on-line.

4. A malicious client B can always abandon any instance of the protocol or intentionally send invalid messages to S (e.g $hc' \neq H(C)$). In such case, A does not receive any credit even though B consumed A's resources (but also B does not receive the file in that instance, as we argue above). This is a denial of service attack against A, and we mitigate it by having S issue a short-lived MAC'ed ticket T_{SA} only to authorized clients. Therefore B can stage this attack against A only for as long as the ticket is valid. If B is identified as misbehaver client, he will not be issued new tickets. In addition, S may charge B for the issuance of tickets T_{SA} effectively preventing B from maliciously expending both A's and S's resources.

5. Two clients B and A could agree to share a file that B is not entitled to receive based on a file access policy and pretend that A is uploading a file to which B has access. In that way, A would get paid, therefore it has incentives to provide the whole file and violate the policy. However, this is problematic because the complaint mechanism can only rule in favor of B, therefore A cannot trust that B will allow A to be rewarded for his service.