Dandelion: Cooperative Content Distribution with Robust Incentives

Michael Sirivianos   Xiaowei Yang   Stanislaw Jarecki
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 robustly 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 [21]. A number of key content producers, (e.g. CBS, Disney, Universal), are now selling films online [3, 4, 7].

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 [6, 15]. 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 [13, 24, 30, 36] 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 [14, 18, 22, 28, 31, 32], most notably BitTorrent [10]. 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 [17, 20, 29] 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 [9]. 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 cost 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. 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. The Dandelion server and the credit bank are logical modules and could be distributed over multiple trusted nodes to improve scalability.

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.

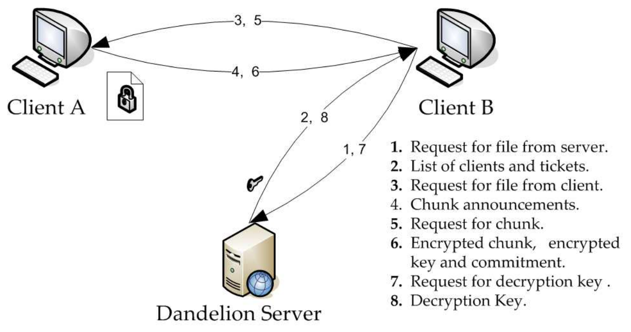
Client A
Client B
Dandelion Server
3, 5
4, 6
2, 8
1, 7
1. Request for file from server.
2. List of clients and tickets.
3. Request for file from client.
4. Chunk announcements.
5. Request for chunk.
6. Encrypted chunk, encrypted key and commitment.
7. Request for decryption key .
8. Decryption Key.

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.

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 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 [10], 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 [10]. *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's incentive mechanism creates a virtual economy that may enable a broad application scenarios. 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.

Dandelion's incentive mechanism is specifically designed for the case in which users maintain paid accounts with the content provider, such as in iTunes. In this case, a client can be rewarded sufficient initial credit to download the content from the server. The content 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. This would motivate a client to upload to others and earn credit.

Dandelion can also be used in the case users do not perform monetary transactions with the content provider, e.g., the distribution of a free Linux Distro. In this scenario, a content provider might not be able to provide monetary incentives for cooperation. We imagine that a creative content provider can use the credit system in multiple ways to motivate cooperation. For instance, the provider may give new clients only a portion *a* of the credit needed to download the complete content. This initial credit enables a new client to download a portion of the content upon joining the Dandelion swarm. Thereafter, a client is incented to upload to others in order to accumulate credit to be used towards downloading the complete content.

A user may attempt to boost its credit by registering multiple Dandelion IDs and claiming to have uploaded to the faked IDs. This will not be a problem in case the user maintains a paid account with the provider, because the user essentially purchases its initial credit, and the net sum in a upload-download transaction between two IDs is zero. However, it may pose a problem in case the user does not pay the content provider, as each ID is given an initial credit.

In this case, the content distributor may require during the registration process a valid mailing address or an internal organization identifier (such as student ID).

2.4 Discouraging Unauthorized Content distribution

If a client participates in cooperative content distribution and is interested in its peer's content, the client can be incented to upload to a peer through the use of a tit-for-tat-based incentive mechanism, regardless of whether the peer is authorized by the content provider. On the other hand, a Dandelion client that is not interested in an unauthorized peer's content is *discouraged* from uploading to that peer. This is because such client has no strong incentive to upload to the unauthorized peer other than the credit he could earn through the use of Dandelion's cryptographic fair-exchange protocol. However, the Dandelion server mediates all transactions that use the fair-exchange protocol. Hence, the server can refrain from rewarding with credit a client that serves unauthorized peers.

For example, selfish seeders have no incentive to facilitate unauthorized content distribution. Our scheme motivates seeders to behave selfishly and not altruistically. That is, seeders are reluctant to waste bandwidth to upload to unauthorized users because they can use their bandwidth to upload to authorized users and earn monetary rewards. Clients are able to verify the legitimacy of requests for service (step 4 in Section 2.2), hence they can avoid wasting bandwidth to serve unauthorized clients. Furthermore, precisely because of this ability, clients can be held liable if they choose to send content 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 performance 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 [23] web server's Asymmetric Multi Process Event Driven Architecture and the Staged Event Driven Architecture [12]. 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. For chunk transmission 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 executed by the thread responsible for the last stage of request processing.

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	~ 2105

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.

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 at the application layer..

We let the Dandelion clients send the following two types of requests to the server and benchmarked the client

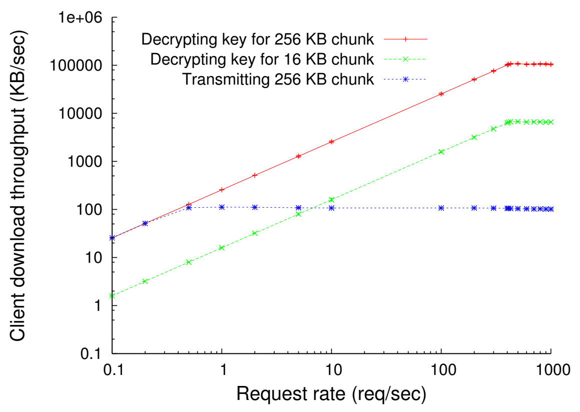

Figure 2: The *y* axis shows the achievable aggregate download throughput of Dandelion clients when the server responds to: a) requests for keys used to decrypt 256 KB encrypted chunks; b) requests for keys used to decrypt 16 KB encrypted chunks; and c) requests for chunks. The *x* axis is the specified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

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 become full, clients would not send new requests at the specified rate due to TCP flow control. 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 ∼420 decryption key requests per second, effectively serving up to ∼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 and induces more load at the server, 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 [10] and Slurpie [28]. 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 [17] 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 [20], 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 [29]. We argue that even if piece-level tit-for-tat [17] is employed, a large view would enable a selfish peer to download large amount of content without needing to reciprocate, taking advantage of the initial barter slack that such a scheme would require.

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 [25]. 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 some BitTorrent deployments, 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 [2, 5]. Our scheme does not require third party infrastructure.

Escrow services in peer to peer networks. Horne et al. [16] 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. [37]. 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 [33] introduces a pairwise credit-based trading mechanism (barter) for peer-to-peer file sharing networks and examines the available peer strategies. Scrivener [11] 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 [34] 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 [26]. Karma uses certified-mail-based [27] 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] http://www.ietf.org/internet-drafts/draft-ietf-tls-rfc2246-bis-13.txt.

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

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

[4] 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.

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

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

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

[8] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. volume 1109, 1996.

[9] 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.

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

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

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

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

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

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

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

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

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

[19] H. Li, A. Clement, E. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. BAR gossip. In *OSDI*, 2006.

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

[21] 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.

[22] V. N. Padmanabhan and K. Sripanidkulchai. The case for cooperative networking. In *IPTPS*, 2002.

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

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

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

[26] 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.

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

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

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

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

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

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

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

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

[35] A. Vlavianos, M. Iliofotou, and M. Faloutsos. Bitos; enhancing bittorrent for supporting streaming applications. In *IEEE Global Internet*, 2006.

[36] 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.

[37] 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 describes in detail Dandelion's cryptographic fair-exchange-based protocol.

A.1.1 System Model

We describe the system model under which Dandelion is designed to operate. We assume three types of clients, which we define as follows:

• *Malicious* clients aim at harming the system. They misbehave as follows: a) they may attempt to cause other clients to be blacklisted or charged for chunks they did not obtain; b) they may attempt to perform a Denial of Service (DoS) attack against the server or selected clients (this attack would involve only protocol messages, as we consider bandwidth or connection flooding attacks outside the scope of this work); and c) they may upload invalid chunks aiming at disrupting the distribution of content.

• *Selfish* (rational) clients share a utility function. This function describes the cost they incur when they upload a chunk to their peers and when they pay virtual currency to download a chunk. It also describes the benefit they gain when they are rewarded in virtual currency for correct chunks they upload and when they obtain chunks they wish to download. A selfish client aims at maximizing its utility. We assume that the content provider prices the content and the accumulated virtual currency of each peer appropriately: the benefit that a selfish client gains from acquiring virtual currency for content it uploads exceeds the cost of utilizing its uplink to upload it.

A selfish client may consider manipulating the system in order to maximize its utility by misbehaving as follows: a) it may not upload chunks to a peer, yet claim credit for them; b) it may upload garbage either on purpose or due to communication failure, and yet claim credit; c) it may obtain chunks from selfish clients, and yet attempt to avoid being charged; d) it may attempt to download content from selfish peers without having sufficient credit; and e) it may attempt to boost its credit by colluding with other clients or by opening multiple Dandelion accounts.

• *Altruistic* clients upload correct content to their peers regardless of the cost they incur and they do not expect to be rewarded.

We assume weak security of the IP network, meaning that a malicious or a selfish client cannot interfere with the routing and forwarding function, and cannot corrupt messages, but it can eavesdrop messages. In addition, we assume that errors can occur during transmissions, resulting in the reception of invalid content.

A.1.2 Setting

By $\langle X \rangle$ we denote the description of an entity or object, e.g. $\langle X \rangle$ 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, *MAC* is a Message Authentication Code (e.g HMAC [8]) and $\langle i \rangle$ refers to a time period. By $\langle i \rangle_X$ we denote the $\langle i \rangle$ at client or server X.

Due to host mobility and NATs, we do not use Internet address (IP or IP/source-port) to associate credit and other persistent protocol information with clients. Instead, each user applies for a Dandelion account and is associated with a persistent ID. The server S matches any client with its authentication information (client ID and password), the file $\langle F \rangle$ it currently downloads or seeds, its credit balance, and the files it can access. The clients and the server maintain loosely synchronized clocks.

Every client A that wishes to join the network must establish a transport layer secure session with the server S, e.g. using TLS [1]. A client sends its ID and password over the secure channel. The server S generates a temporary shared secret key K_{SA} with A. K_{SA} is efficiently computed as $K_{SA} = H(K_S, \langle A \rangle, \langle i \rangle)$. K_{SA} is also sent over the secure channel. The server initiates re-establishment of shared keys with the clients upon $\langle i \rangle$ change to prevent attackers from: a) inferring the key by examining the encrypted content and the MACs used by the protocol; and b) replaying previously sent messages that were signed using K_{SA}, while allowing the reuse of message sequence numbers once the numbers reach a high threshold. S tolerates some lag in the $\langle i \rangle$ assumed by a client.

The rest of the messages that are exchanged between the server and the clients are sent over an insecure connection (e.g. plain TCP), which must originate from the same IP as the secure session. Similarly, all messages between clients are sent over an insecure connection.

A.1.3 Protocol Description

To provide robust incentives for cooperation, Dandelion employs a cryptographic fair-exchange mechanism. The server serves as the trusted third party (TTP) 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 the decryption key 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.

When a 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 sends garbage to B, A cannot deny it. In addition, B is prevented from falsely claiming that A has sent it garbage.

Our fair-exchange protocol involves only efficient symmetric cryptographic operations. Each client B exchanges only short messages with the server. Each such message includes a sequence number and a digital signature. The signature is computed as the MAC of the message, which includes a sequence number, keyed with the secret key K_{SB} that B shares with the server. Each time a client or the server receive a message from each other, they check whether the sequence number succeeds the sequence number of the previously received message and whether the MAC-generated signature verifies. If either of the two conditions is not satisfied, the message is discarded. The sequence number is reset when time period $\langle i \rangle$ changes. The following description omits the sequence number and the digital signature. Figure 1 depicts the message flow in our system.

Step 1: The protocol starts with the client B sending a request for the file $\langle F \rangle$ to S.

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

Step 2: If B has access to $\langle F \rangle$, 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 ID of the client, also contains the client's inbound Internet address. 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. 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 and S) and allow B to request chunks of file $\langle F \rangle$ from client 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 ensure file integrity, S also sends the SHA-1 hash $h_{\langle c \rangle} = H(c)$ for all chunks c of the file $\langle F \rangle$.

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

Step 3: The client B forwards this request to each $A \in \langle A \rangle_{\text{list}}$.
$B \longrightarrow A$:[client file request] $T_{SA}, \langle F \rangle, \text{ts}, \langle i \rangle_S$

Step 4: If *current-time* $\leq ts + T$ and T_{SA} is not in A's cache, A verifies if $T_{SA} = MAC_{K_{SA}}[\langle A \rangle, \langle B \rangle, \langle F \rangle, \text{ts}]$. The purpose of this check is to mitigate DoS attacks against A; it allows A to filter out requests from clients that are not authorized to retrieve the content or from clients that became blacklisted. As long as B remains connected to A, it periodically renews its T_{SA} tickets by requesting them from S. 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 replies with the message described below, which contains a list of all the chunks A owns, $\langle c \rangle_{\text{list}}$. In addition, A periodically reports to B newly acquired chunks as long as the timestamp ts is fresh. B also does so in separate *chunk announcement* messages.

$A \longrightarrow B$:[chunk announcement] $\langle c \rangle_{\text{list}}$

Step 5: B and A determine which chunks to download from each other according to a chunk selection policy; BitTorrent's locally-rarest-first is suitable for static content dissemination, while other policies [19, 35] are more appropriate for streaming content. A can request chunks from B, after it requests and retrieves T_{SB} from S. B sends a request for the missing chunk c to A.

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

Step 6: B's *chunk requests* are served by A as long as the timestamp is fresh, and T_{SA} is cached or T_{SA} verifies. If A is altruistic, it sends the chunk c to B in plaintext and the per-chunk transaction ends here. Otherwise, A encrypts c using a symmetric encryption algorithm Enc,as $C = Enc^{\text{iv}_{\langle c \rangle}}_{k_{\langle c \rangle}}(c)$, where $k_{\langle c \rangle}$ is a randomly generated key that is distinct for each chunk, and $\text{iv}_{\langle c \rangle}$ is the randomly generated encryption Initialization Vector (IV). A encrypts the random key with K_{SA}, as $e = Enc^{\text{iv}_{\text{SA}}}_{K_{SA}}(k_{\langle c \rangle}, \text{iv}_{\langle c \rangle})$. Next, A hashes the ciphertext C as $H(C)$. Subsequently, it computes its commitment to the encrypted chunk and the encrypted key as $T_{AS} = MAC_{K_{SA}}[\langle A \rangle, \langle B \rangle, \langle F \rangle, \langle c \rangle, e, H(C), \text{ts}]$ and sends the following to B.

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

Step 7: To retrieve $(k_{\langle c \rangle}, \text{iv}_{\langle c \rangle})$, B needs to request it from the server. B computes its own hash over the received ciphertext C' and it forwards the following to S.

$B \longrightarrow S$:[decryption key request] $\langle A \rangle, \langle F \rangle, \langle c \rangle, e, H(C')$, $\text{ts}, T_{AS}, \langle i \rangle_A$

Step 8: If timestamp ts is fresh enough, and $\langle i \rangle_A$ is not too much off, S checks if $T_{AS} = MAC_{K_{SA}}[\langle A \rangle, \langle B \rangle, \langle F \rangle, \langle c \rangle, e, H(C'), \text{ts}]$, where key K_{SA} is a cached value or a value computed using K_S, $\langle A \rangle$, and $\langle i \rangle_A$. The timestamp ts freshness requirement forces B to expedite paying for decrypting the encrypted chunks. This fact allows A to promply acquire credit for its service. The ticket T_{AS} verification may fail either because $C' \neq C$ due to transmission error in step (6) or because A or B are misbehaving. Since S is unable to determine which is the case, it does not punish either clients and does not update their credit. It does not send the decryption key to B, but it still notifies B of the discrepancy. In this case, B is expected to disconnect from A and blacklist it in case A repeatedly sends invalid messages. If B keeps sending invalid decryption key requests, S penalizes him. If the verification succeeds, S checks whether B has sufficient credit to purchase the chunk c. It also checks again whether B has access to the file $\langle F \rangle$. If B is approved, it charges B and rewards A with Δ_c credit units. Subsequently, S decrypts $(k'_{\langle c \rangle}, \text{iv}'_{\langle c \rangle}) = Dec_{K_{SA}}(e)$, and sends them to B.

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

Next, we explain the complaint mechanism.

Step 9: B uses $(k'_{\langle c \rangle}, \text{iv}'_{\langle c \rangle})$ to decrypt the chunk as $c' = Dec^{\text{iv}'_{\langle c \rangle}}_{k'_{\langle c \rangle}}(C)$. If the decryption fails or if $H(c') \neq h_{\langle c \rangle}$ (step (2)), B complains to S by sending the following message.

$B \longrightarrow S$:[complaint] $\langle A \rangle, \langle F \rangle, \langle c \rangle, T_{AS}, e, H(C'), \text{ts}, \langle i \rangle_A$

S ignores this message if *current-time* $> ts + T'$, where $T' > T$. $T' - T$ should be greater than the time needed for B to receive a decryption key response, decrypt the chunk and send a complaint to the server. This condition ensures that a misbehaving A cannot avoid having complaints ruled against it, even if A ensures that the time elapsed between the moment A commits to the encrypted chunk and the moment the encrypted chunk is received by B is slightly less than T.

S also ignores the complaint message if a complaint for the same A and c is in a cache of recent complaints that S maintains for each client B. Complaints are evicted from this cache once *current-time* $> ts + T'$. If $T_{AS} \neq MAC_{K_{SA}}[\langle A \rangle, \langle B \rangle, \langle F \rangle, \langle c \rangle, e, H(C'), \text{ts}]$, S punishes B. This is because S has already notified B in step (7) that T_{AS} is invalid. If T_{AS} verifies, S caches this complaint, recomputes K_{SA} as before, decrypts $(k'_{\langle c \rangle}, \text{iv}'_{\langle c \rangle}) = Dec_{K_{SA}}(e)$ once again, retrieves c from its storage, and encrypts c himself using the above key and IV vector, $C'' = Enc^{\text{iv}'_{\langle c \rangle}}_{k'_{\langle c \rangle}}(c)$. If the hash of the ciphertext $H(C'')$ is equal to the value $H(C')$ that B sent to S, S decides that A has acted correctly and B's complaint is unjustified. Subsequently, S drops the complaint request and blacklists B. It also notifies A, which disconnects from B and blacklists it. Otherwise, if $H(C'') \neq H(C')$, S decides that B was cheated by A, removes A from its set of active clients, blacklists A, and revokes the corresponding credit charge on B. Similarly, B disconnects from A and blacklists it.

The server disconnects from a blacklisted client E, marks it as blacklisted in the credit file and denies access to E if it attempts to login. Future complaints concerning a blacklisted client E and for which T_{ES} verifies, are ruled against E without further processing.

Since a verdict on a complaint can adversely affect a client, each client needs to ensure that the commitments it generates are correct even in the rare case of a disk read error. Therefore, a client always verifies the read chunk against its hash before it encrypts the chunk and generates its commitment.

A.2 Security Analysis

We claim the following security properties of our design:

Lemma 4.1 If the server S charges a client B Δ_c credit units for a chunk c received from a selfish client A, B must have received the correct c, regardless of the actions taken by A.

Proof 4.1 B gets charged only if the ticket T_{AS} that S gets in steps (6) and (8) verifies and if $H(C) = H(C')$. Since $H(C)$ is a second pre-image resistant cryptographic hash that B computes itself on C received from A, $C = C'$. Furthermore, since the same k, iv pair is used by A to encrypt c into C' and by B to decrypt C into c', $C = C'$ implies that $c' = c$.

Lemma 4.2 If a selfish client A always encrypts chunk c anew when servicing a request and if B gets correct c from A, then A is awarded Δ_c credit units from S, and B is charged $Delta_c$ credit units from S.

Proof 4.2 If A encrypts c using a one-time key $k_{\langle c \rangle}, \text{iv}_{\langle c \rangle}$, B sees $k_{\langle c \rangle}, \text{iv}_{\langle c \rangle}$ only in the encrypted form e. The only way E can retrieve the encrypted chunk is by retrieving the shared secret key K_{SA}. However, this key was transmitted over the secure session between A and the server S. Therefore, the only way for B to retrieve it 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 a complaint, which includes T_{AS} and $H(C')$ such that. $T_{AS} = MAC_{K_{SA}}[\langle A \rangle, \langle B \rangle, \langle F \rangle, \langle c \rangle, e, H(C'), \text{ts}]$, while $H(C') \neq H(C'')$ where C'' is computed by S in step (9). However, since we consider this attack only against a selfish A, the T_{AS} value will verify only if all the values it includes are the ones that A sent to B, and if hash $H(C')$ is correctly computed on the transferred ciphertext that is C. However, if this is the case, S will decrypt e to the same $k_{\langle c \rangle}, \text{iv}_{\langle c \rangle}$ pair that A used, hence S's encryption C'' will be the same as the C that A computed. Consequently, $H(C')$ will be equal to $H(C'')$, hence B is not able to reverse its charge.

Lemma 4.3 A selfish or a malicious client cannot assume another authorized client's A identity and issue messages under A, aiming at obtaining service at the expense of A, charging A for service it did not obtain or causing A to be blacklisted. In addition, it cannot issue a valid T_{SA} for an invalid chunk that it sends to a client B and cause B to produce a *complaint* message that would result in a verdict against A.

Proof 4.3 The only way a misbehaving client can be successful in such attack is by obtaining the user authentication information or the shared secret key K_{SA}. However user authentication, and the transmission of the shared secret key K_{SA} is performed over the secure session between A and the server S.

Lemma 4.4 A malicious client cannot replay previously sent valid requests to the server or generate *decryption key chunk requests* or *complaints* under A's ID, aiming at A being charged for service it did not obtain or being blacklisted because of invalid or duplicate complaints.

Proof 4.4 All messages exchanged between a client A and the server are digitally signed with the shared secret key K_{SA} and include sequence numbers. Both the client and the server store the last sequence number seen by each other and the sequence numbers are reset upon $\langle i \rangle$ change. Thus, a malicious client cannot forge the source of the request, neither it can resent a request that has already been received.

Observation 4.5 A client cannot download chunks from a selfish peer if it does not have sufficient credit. Since the server mediates each exchange the credit balance is always checked before the client retrieves the decryption key for a chunk.

Observation 4.6 To maintain an efficient content distribution pipeline, a client needs to relay a received chunk to its peers as soon as it receives it. However, the chunk may be invalid due to communication errors or due to peer misbehavior. The performance of the system would be severely degraded if peers wasted bandwidth to relay invalid content. To address this issue, Dandelion clients send a decryption key request to the server immediately upon receiving the encrypted chunk. This design choice enables clients to promptly retrieve the chunk in its non-encrypted form. Thus, they can verify the chunk's integrity prior to uploading the chunk to their peers.

Observation 4.7 A malicious client cannot DoS attack the server by sending invalid content to other clients or repeatedly sending invalid complaints aiming at causing the server to perform the relatively expensive complaint validation. This is because it becomes blacklisted by both the server and its peers the moment the invalid complaint is ruled against it. In addition, a malicious client cannot attack the server by sending valid signed messages with redundant valid complaints. Our protocol detects duplicate complaints through the use of timestamps and caching of recent complaints.

Observation 4.8 A malicious client E can always abandon any instance of the protocol. In such case, E does not receive any credit, as argued in Lemmas 4.1 to 4.3, even though E may have consumed A's resources. This is a denial of service attack against A. Note that this attack would require that the attacker expends resources proportional to the resources of the victim, therefore it is not particularly practical. Nevertheless, we prevent blacklisted clients or clients that do not maintain paid accounts with the content provider from launching such attack by having S issue only to authorized clients a short-lived ticket T_{SA}. T_{SA} is checked for validity by A (steps 4 and 6 in Section A.1.3). In addition, S may charge an authorized E for the issuance of tickets T_{SA} effectively deterring E from maliciously expending both A's and the server's resources.

Owing to properties 4.1, 4.2, 4.3 and 4.5, and given that the content provider employs appropriate pricing schemes, Dandelion ensures that selfish (rational) clients increase their utility when they upload correct chunks and obtain virtual currency, while misbehaving clients cannot increase their utility. Consequently, Dandelion entices selfish clients to upload to their peers, resulting in a Nash equilibrium of cooperation.