Dandelion: Cooperative Content Distribution with Robust Incentives

Michael Sirivianos Jong Han Park Xiaowei Yang Stanislaw Jarecki
Department of Computer Science
University of California, Irvine
{*msirivia,jonghanp,xwy,stasio*}*@ics.uci.edu*

Abstract

Content distribution via the Internet is becoming increasingly popular. To be cost-effective, commercial content providers are considering the use of peer-to-peer (P2P) protocols such as BitTorrent to save on bandwidth costs and to handle peak demands. However, when an online content provider uses a P2P protocol, it faces a crucial issue: how to incentivize its clients to upload to their peers.

This paper presents Dandelion, a system designed to address this issue in the case of paid content distribution. Unlike previous solutions, most notably BitTorrent, Dandelion provides robust (provably non-manipulable) incentives for clients to upload to others. In addition, unlike systems with tit-for-tat-based incentives, a client is motivated to upload to its peers even if the peers do not have content that interests the client. A client that honestly uploads to its peers is rewarded with credit, which can be redeemed for various types of rewards, such as discounts on paid content.

In designing Dandelion, we trade scalability for the ability to provide robust incentives. The evaluation of our prototype system on PlanetLab demonstrates the viability of our approach. A Dandelion server that runs on commodity hardware with a moderate access link is capable of supporting up to a few thousand clients. These clients can download content at rates comparable to those of BitTorrent clients.

1 Introduction

Content distribution via the Internet is becoming increasingly popular among the entertainment 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 [10]. A number of key content producers, such as Universal, are now launching download to own services [15]. However, the increasing demand for digital content is overwhelming the infrastructure of online content providers [13].

An attractive approach for commercial online content distribution is the use of peer-to-peer (P2P) protocols. This approach does not require a content provider to overprovision its bandwidth to handle peak demands, nor does it require the provider to purchase service from a third-party such as Akamai. Instead, a P2P protocol such as BitTorrent [26] harnesses its clients' bandwidth for file distribution, and saves the bandwidth and computing resources of a content provider. Leading content providers such as Warner Bros [16] and 20th Century Fox [11] have now partnered with BitTorrent, Inc. EMI [12] has announced a plan to launch a P2P music distribution service. This recent trend indicates that P2P protocols enable a site to cost-effectively distribute content.

When an online content provider uses a peer-to-peer protocol, it faces a crucial issue: how to motivate clients that possess content to upload to others. This issue is of paramount importance because the performance of a P2P network is highly dependent on the users' willingness to contribute their uplink bandwidth. However, selfish (rational) users tend not to share their bandwidth without external incentives [36]. Although the popular BitTorrent protocol has incorporated the rate-based tit-for-tat incentive mechanism for users to upload static content, this mechanism bears two weaknesses. First, it does not encourage clients to seed, i.e. to upload to other peers after completing the file download. Second, it is vulnerable to manipulation [37, 43, 44, 48, 49], allowing modified clients to free-ride and still achieve a better downloading rate than compliant clients (Section 2.2).

The purpose of this work is to explore the design space of a P2P content distribution protocol that addresses this issue. We present the design and implementation of Dandelion, a cooperative paid content distribution protocol that uses non-manipulable virtual-currency-based incentives to encourage uploading and to address free-riding.

Our protocol guarantees strict fair exchange of content uploads for virtual currency (credit). A client cannot download content from *selfish* peers (i.e. peers that do

not upload unless they expect to be rewarded) without paying credit, neither it can obtain credit for uploads it did not perform. This protocol property provides robust incentives for selfish peers to contribute their bandwidth in the following two ways. First, credit can be redeemed at a content provider for a discount on the content, or for other types of monetary awards. Given appropriate pricing schemes, a selfish client is motivated to serve content to its peers regardless of whether its peers possess content that interests it. Second, the protocol prevents free-riding, because, provably, the only way a client can obtain valid content from selfish peers or can earn credit is by paying credit or uploading valid content, respectively. Hence, we believe that our protocol can increase the aggregate upload bandwidth of a P2P content distribution system and improve downloading times.

The use of virtual currency for incentives has been proposed in several P2P content distribution systems [3, 7, 8, 17, 29, 51, 52], but a key challenge, how to make the virtual-currency-based system efficient and practical while robust to manipulation, is left unaddressed (Sections 2.1 and 2.4). We address this challenge based on the insight that in the problem domain of online content distribution, the content provider itself is a trusted third party and can mediate the content exchange between its clients. Based on this observation, we design a protocol in which clients exchange data for credit and the server mediates this exchange. The server uses only efficient symmetric cryptography on critical data paths and sends only short messages to its clients.

Our work makes the following contributions:

1) An efficient cryptographic fair exchange scheme for trading data uploads for virtual currency, which is suitable for P2P content distribution. Our scheme is based on symmetric key cryptography, and is provably robust to client cheating. A client that does not upload or uploads garbage to its peers cannot claim credit. A client cannot download correct content from selfish peers without the client being charged and the peers rewarded.

2) The design and implementation of Dandelion. To the best of our knowledge, Dandelion is the first implemented P2P static content distribution protocol that uses symmetric cryptography in order to provide robust incentives for clients to upload paid content to their peers. Our system's evaluation on PlanetLab [24] identifies the scalability limits of our incentive mechanism and demonstrates the plausibility of our approach.

In this paper, we use a BitTorrent-like terminology. A *seeder* refers to a client that uploads to its peers despite not being interested in the content being distributed (e.g. a client that uploads although it has completed its file download). A *leecher* refers to a client that is interested in the content being distributed (e.g. a client that has not completed its file download). A *free-rider* refers to a client that only downloads from others but does not upload. A *swarm* refers to all clients that actively participate in the protocol for a given content item.

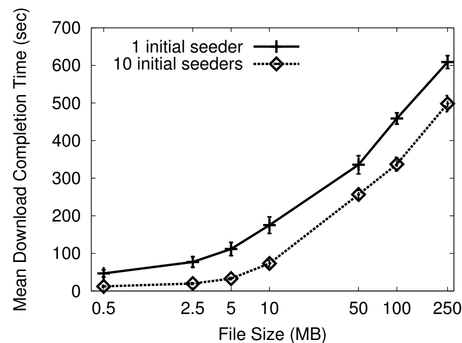

Figure 1: **Mean download completion times of ∼200 CTorrent 1.3.4 leechers as a function of file size, for 1 initial seeder and 10 initial seeders. The mean file download completion times are extracted over 10 runs. Error bars correspond to 95% confidence intervals.**

The rest of this paper is organized as follows. Section 2 describes existing incentive mechanisms in P2P protocols, and cryptographic fair exchange schemes. Section 3 describes the design of Dandelion. Section 4 analyzes the security of our design. Sections 5 and 6 present our system's implementation and experimental evaluation, respectively. We conclude in Section 7.

2 Background and Related Work

In this section we motivate the design of Dandelion by describing existing P2P content distribution incentive mechanisms and their weaknesses. In addition, we discuss previous work on cryptographic fair exchange.

2.1 Impact of Seeding

The popular BitTorrent protocol employs the rate-based tit-for-tat incentive mechanism [26]. A client unchokes (i.e. uploads to) at most four to ten clients for a given file, in parallel. Most of the unchoked peers are the peers that upload useful parts of the file to the client at the fastest rates and are interested in the client's content. The client also *optimistically* unchokes one or two peers that are not among the fastest uploaders, in expectation of future reciprocation. The list of unchoked peers is typically revised every 10 seconds.

This mechanism mitigates free-riding but does not provide explicit incentives for seeding. Although several BitTorrent deployments rely on clients to honestly report their uploading history [17], and use this history to decide which clients can join a swarm, practice has shown that clients can fake their upload history reports [4].

In contrast, Dandelion's non-manipulable and centrally maintained virtual currency enables a content distributor to reliably keep record of the amount of content a selfish client has uploaded to its peers. The distributor can use this record to provide robust incentives for a selfish client to upload to its peers regardless of whether the peers have content that interests the client.

Seeders can substantially improve download completion times, because they increase the file availability and the aggregate upload bandwidth. Figure 1 shows the impact of seeders. We run two BitTorrent experiments on PlanetLab, with one and ten initial seeders, respectively. Initial seeders are clients that have the complete file prior to the start of the distribution. In each experiment, we run ∼200 CTorrent 1.3.4 [2] leechers on distinct PlanetLab nodes to simultaneously download a file. Upon completion of their download, leechers remain online seeding the file. As can be seen, the mean file download completion time decreases considerably when there are ten initial seeders, especially for small files of a few MB.

2.2 Free-riding in BitTorrent

A general observation is that since BitTorrent's tit-for-tat incentives reward cooperative clients with improved download times, clients are always incented to upload. Therefore, free-riding should not be an issue in BitTorrent networks. This observation relies on the assumption that users aim only at maximizing their download rates. However, in practice, several BitTorrent users can be reluctant to upload even if uploading improves their download times. For example, users with access providers that impose quotas on outgoing traffic or users with limited uplink bandwidth (e.g. 1.5Mbps/128Kbps ADSL) may wish to save their uplink for other critical tasks.

Considering the tradeoff between performance and susceptibility to free-riding [31], BitTorrent purposely does not implement a strict tit-for-tat (TFT) strategy. In particular, it employs rate-based instead of chunk-level TFT, and BitTorrent clients optimistically unchoke peers for relatively long periods of time (10 to 30 seconds). Furthemore, BitTorrent seeders select peers to upload to regardless of whether those peers upload to others.

Based on the above observations and previous work on BitTorrent exploitation [37, 43, 48], in [49], we modify a CTorrent-1.3.4 client to employ the "large view" exploit to free-ride. The client obtains a larger than normal view of the swarm, either by repeatedly requesting partial views from the BitTorrent tracker or by exchanging views with its peers [6, 9]. Subsequently, it connects to all peers in its view, while it does not upload any content. Using this exploit in a sufficiently large swarm, a modified client can substantially increase the frequency with which it becomes optimistically unchoked, comparing to a compliant client, which typically connects to 50-100 peers. It can also find more seeders, which do not employ tit-for-tat.

In particular, we show that our modified free-rider client is able to download faster than its tit-for-tat compliant counterpart in 12 out of 15 randomly selected public torrents, for file sizes between 500MB to 2 GB and swarm sizes of 50 to 1000 peers. We also experiment with PlanetLab residing swarms that comprise of ∼300 leechers that are rate-limited at 30KB/sec and one initial seeder that is rate-limited at 120KB/sec. When compliant clients comprise 90% of the PlanetLab-residing swarm, free-riders download faster than compliant clients in their swarm and slightly worse than compliant clients in a swarm with no free-riders.

The same weakness of BitTorrent's incentives is experimentally demonstrated in a recent work by Locher et al. [44], which was almost concurrent with ours.

Drawing from the above observations, we believe that the "large view" exploit has the potential to be widely adopted and could lead to system-wide performance degradation in BitTorrent swarms. Dandelion explicitly addresses this issue, because its provably non-manipulable incentives enable a content distributor to reliably track the amount of content a client has downloaded from selfish peers, and charge the client accordingly.

2.3 Pairwise Currency as Incentives

In P2P content distribution protocols that employ pairwise virtual currency (credit) for incentives, clients maintain distinct credit balances for each of their peers. In this context, credit refers to any metric of a peer's cooperativeness.

An eMule [7] client rewards cooperative peers by reducing the time the peers have to wait until they are served by the client. Swift [51] introduces a pairwise credit-based trading mechanism for peer-to-peer file sharing networks and examines the available peer strategies. In [37], the authors suggest tackling free-riding in BitTorrent by employing chunk-level tit-for-tat, which is similar to pairwise credit incentives. These pairwise credit-based incentive mechanisms bear weaknesses that are similar to the ones of rate-based tit-for-tat: a) they provide no explicit incentives for seeding; and b) they can be manipulated by free-riders that obtain a "large view" of the network, and initiate short-lived sessions with numerous peers to exploit the initial offers in pairwise transactions.

Scrivener [29] combines pairwise credit balances with a transitive trading mechanism. Its incentive mechanism is based on the premise that a client remains perpetually interested in exchanging the earned pairwise credit for content downloads from the same network. Unlike

Scrivener, credit earned by Dandelion clients can be converted into monetary rewards, providing strong incentives for clients to upload even if the network ceases to offer content that interests the client.

MojoNation [8] used a combination of pairwise balances and tokens that can be cashed in a central broker. When the debt during pairwise transactions exceeds a specified threshold, the side with the negative balance transfers a credit token to the other by contacting a broker. Since MojoNation does not provide strong fair exchange guarantees of content uploads for credit, it can be manipulated in a way similar to the "large view" exploit.

Keidar et al. [38] present the design of a P2P multicast protocol, which is formally proven to enforce cooperation among selfish leechers. To prove cooperation, the authors assumed that selfish leechers abide by a predetermined strategy, which specifies how many peers a leecher can have. However, the recent work on BitTorrent exploitation [44, 49], which has partly motivated our system's design, has demonstrated that this assumption may be too restrictive.

BAR Gossip [41] is suitable for P2P streaming of live content. Owing to its cryptographic exchange mechanism it is robust to clients that attempt to free-ride. Since BAR Gossip is designed for P2P streaming, it does not need to provide incentives for seeding. Therefore, it ensures the fair exchange of content uploads between clients that are interested in the same live broadcast. On the other hand, Dandelion, which needs to incent seeding for static content distribution or video on demand, guarantees fair exchange of content uploads for virtual currency.

2.4 Global Currency as Incentives

It has been widely proposed to use global virtual currency to provide incentives in P2P content distribution systems. This is the basis of the incentive mechanism employed by Dandelion: for each client, the system maintains a credit balance, which is used to track the bandwidth that the client has contributed to the network.

Karma [52] employs a global credit bank and certified-mail-based [47] fair exchange of content for reception proofs. It distributes credit management among multiple nodes. Karma's distributed credit management improves scalability. However, it does not guarantee the integrity of the global currency when the majority of the nodes that comprise the distributed credit bank are malicious or in a highly dynamic network. In contrast, Dandelion's centrally maintained global currency is non-manipulable by clients, enabling a content provider to incent client cooperation by offering monetary rewards.

Horne et al. [35] proposed an encryption- and erasure-code-based fair exchange scheme for exchange of content for proofs of service, but did not provide an experimental evaluation. Their scheme detects cheating with probabilistic guarantees, whereas Dandelion deterministically detects and punishes cheaters.

Li et al. [42] proposed a scheme for incentives in P2P environments that uses fair exchange of proof of service with chunks of content. The selfish client encrypts a chunk and sends it to its peer, the peer responds with a public-key cryptographic proof of service, and the client completes the transaction by sending the decryption key. A trusted third party (TTP) is involved only in the following cases: a) the selfish client presents the proofs of service to obtain credit; b) the peer complains for receiving an invalid chunk; and c) the peer complains for not receiving the decryption key from the selfish client. However, unless the server incurs the high cost of frequently renewing the public key certificates of each client, the credit system is vulnerable to clients that obtain content from selfish peers, despite those clients not having sufficient credit. In contrast, in Dandelion, the TTP mediates every chunk exchange, effectively preventing a client from obtaining any chunks from selfish peers without having sufficient credit.

PPay [54] and WhoPay [53] are recent *micropayments* proposals that employ public key cryptography and are designed for the P2P content distribution case. These systems do not guarantee fair exchange of content for payment. Free-riders may establish short-lived sessions to many peers, download portions of content from them or obtain payments, and thereby obtain a substantial amount of content or credit without paying or uploading.

2.5 Cryptographic Fair Exchange

There are two main classes of solutions for the classic cryptographic fair exchange problem. One uses simultaneous exchange by interleaving the sending of the message with the sending of the receipt [22, 25, 27, 30, 46]. These protocols rely on the assumption of equal computational and bandwidth capacity, which does not suit the heterogeneous P2P setting.

The other class relies on the use of a trusted [18, 19, 55, 56] or semi-trusted [32, 33] third party (TTP). The main differences of our scheme are as follows: 1) In [18, 19, 56] the TTP cannot decide whether a party has misbehaved, but can only complete the transaction itself if presented with proof that the parties initially intended to perform the transaction. They assume that the cost of sending the data is small and can be repeated by the TTP. However in Dandelion, transmission of data is the most expensive resource and our scheme aims at the fair exchange of this resource; 2) Unlike [32] and [33], our scheme does not rely on untrusted clients to become semi-TTP; 3) Unlike [55], our scheme does not use public key cryptography for encryption and for committing to messages, and only requires one client rather than two

to contact the TTP for each 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.

3 Design

In this section we describe the system model and the design of Dandelion.

3.1 Overview

Our design is based on the premise that a low cost server does not have sufficient network I/O resources to directly serve content to its clients under overload [14, 34]. It may however, have sufficient CPU, memory, and memory/disk/network I/O resources to execute many symmetric cryptography operations, to maintain TCP connection and protocol state for many clients, to access its client's protocol state, and to receive and send short messages. However, CPU, memory and I/O are still limited resources. Therefore we aim at making the design as efficient as possible.

Under normal workload, A Dandelion server behaves similar to a web/ftp, streaming or video on demand server, i.e. it directly serves content to its clients. When a server is overloaded, it enters a *peer-serving* mode. Upon receiving a request, the server redirects the client to other clients that are able to serve the requests for content. In the peer-serving mode, a Dandelion system is reminiscent of BitTorrent, in the sense that a server splits content into verifiable *chunks*, and clients exchange carefully selected chunks. As is the case with BitTorrent, the content is split into multiple chunks in order to enable clients to upload as soon as they receive and verify a small portion of the content. It is also split in order to increase the entropy of content in the network, facilitating chunk exchanges among peers. We discuss the tradeoffs in selecting a chunk size in the case of static content distribution in Section 6.3.1.

However, our protocol uses a different incentive mechanism. The server maintains a virtual economy and associates each client with its credit balance. It entices selfish clients to upload to others by explicitly rewarding them with virtual credit, while it charges clients that download content from selfish peers.

3.2 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 a peer's accumulated virtual currency 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 credit system in order to maximize its utility by misbehaving as follows: a) it may not upload chunks to a peer, and 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 communication errors may occur during message transmissions.

In the rest of this section we describe the design of Dandelion, which explicitly addresses the challenges posed by selfish and malicious clients, as well as the challenges posed by the communication channel.

3.3 Credit Management

Dandelion's incentive mechanism creates a virtual economy, which enables a variety of application scenarios. A client spends $\Delta_c > 0$ credit units for each chunk it downloads from a selfish client and a selfish client earns $\Delta_r > 0$ credit units for each chunk it uploads to a client. A client can acquire a chunk only if its credit is greater than Δ_c. We set $\Delta_c = \Delta_r$, so that two colluding clients cannot increase the sum of their credit, by falsely claiming that they upload to each other.

Our protocol is intended for the case in which users maintain paid accounts with the content provider, such as in iTunes. A client is awarded sufficient initial credit to download the complete paid content from its peers. 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 incents a client to upload to others and earn credit. A user cannot boost its credit by presenting multiple IDs (Sybil attack [28]) and claiming to have uploaded to some of its registered IDs. This is because each user maintains an authenticated paid account with the provider. The user essentially purchases its initial credit, and the net sum in an upload-download transaction between any two IDs is zero.

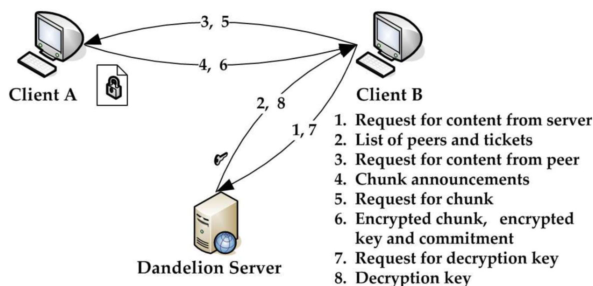

Figure 2: **The Dandelion protocol. The numbers on the arrows correspond to the listed protocol messages and the steps listed in Section 3.4.2. The messages are sent in the order they are numbered.**

3.4 Robust Incentives

This section describes Dandelion's cryptographic fair-exchange-based protocol.

3.4.1 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 the server S's master secret key, H is a cryptographic hash function such as SHA-1, MAC is a Message Authentication Code such as HMAC [20], and p refers to a time period. By p_X we denote p 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 associates each client with its authentication information (client ID and password), the content (e.g. a file) $\langle F \rangle$ it currently downloads or seeds, its credit balance, and the content 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 secret key and symmetric encryption initialization vector pair, denoted K_{SA}, which is shared with A. K_{SA} is efficiently computed as $K_{SA} = (H(K_S, \langle A \rangle, p, 0), H(K_S, \langle A \rangle, p, 1))$. K_{SA} is also sent over the secure channel. This key is used both for symmetric encryption and for computing a MAC. For MAC computation, we use only the secret key portion of K_{SA}. The rest of the messages that are exchanged between the server and the clients are sent over an insecure communication channel (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 communication channel.

Each client A exchanges only short messages with the server. To prevent forgery of the message source and replay attacks, and to ensure the integrity of the message, each message includes a sequence number and a digital signature. The signature is computed as the MAC of the message, keyed with the secret key K_{SA} that A 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 p changes.

The server initiates re-establishment of shared keys K_{SA} with the clients upon p change in order to: a) prevent attackers from inferring K_{SA} by examining the encrypted content and the MACs used by the protocol; and b) allow the reuse of message sequence numbers once these numbers reach a high threshold, while preventing attackers from replaying previously signed and sent messages. The server tolerates some lag in the p assumed by a client.

3.4.2 Protocol Description

To provide robust incentives for cooperation under the model described in Section 3.2, Dandelion employs a cryptographic fair-exchange mechanism. Our fair-exchange protocol involves only efficient symmetric cryptographic operations. The server acts 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 content to B, and charges B for downloading content.

When a client A sends invalid content to B, B can determine 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 that it did. In addition, B is prevented from falsely claiming that A has sent it garbage.

The following description omits the sequence number and the signature in the messages between clients and the server. Figure 2 depicts the message flow in our protocol.

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

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

Step 2: If B has access to $\langle F \rangle$, S chooses a short list of clients $\langle A \rangle_{\text{list}}$, which are currently in the swarm for $\langle F \rangle$. The policy with which the server selects the list $\langle A \rangle_{\text{list}}$ depends on the specifics of the content distribution system. 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 $T_{SA} = MAC_{K_{SA}}[\langle A \rangle, \langle B \rangle, \langle F \rangle, t]$ to B. t 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_{peer} and allow B to request chunks of the content $\langle F \rangle$ from client A. When T_{SA} expires and B still wishes to download from A, it requests a new T_{SA} from S.

To ensure integrity in the case of static content distribution or video on demand, S also sends to B the SHA-1 hash $H(c)$ for all chunks c of $\langle F \rangle$. For the case of live streaming content, the content provider augments the chunks it generates with his public key signature on their hash and ID, as $sign(H(c), \langle c \rangle)$. Clients append this signature to all the chunks they upload.

$S \longrightarrow B$:[content response] $T_{SA}, \langle A \rangle_{\text{list}}, H(c)_{\text{list}}, \langle F \rangle, t, p_S$

Step 3: The client B forwards this request to each $A \in \langle A \rangle_{\text{list}}$.

$B \longrightarrow A$:[content request] $T_{SA}, \langle F \rangle, t, p_S$

Step 4: If *current-time* $\leq t + T_{peer}$ and T_{SA} is not in A's cache, A verifies if $T_{SA} = MAC_{K_{SA}}[\langle A \rangle, \langle B \rangle, \langle F \rangle, t]$. 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. If the verification fails, A drops this request. Also, if p_S is greater than A's current epoch p_A, A learns that it should renew its key with S soon. Otherwise, A caches T_{SA} and periodically sends the chunk announcement message described below, for as long as the timestamp t is fresh. This message contains a list of chunks that A owns, $\langle c \rangle_{\text{list}}$. B also does so in separate chunk announcement messages. The specifics of which chunks are announced and how frequently depend on the type of content distribution. For example, in the case of static content distribution, the initial chunk announcement would contain the IDs of all the chunks A owns, while the periodically sent announcement would contain the IDs of newly acquired chunks.

$A \longrightarrow B$:[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 for streaming content or video on demand other policies are appropriate [23, 41]. 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] $T_{SA}, \langle F \rangle, \langle c \rangle, t, p_S$

Step 6: B's chunk requests are served by A as long as the timestamp t 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 $C = Enc_{k_{\langle c \rangle}}(c)$. $k_{\langle c \rangle}$ is a random secret key and random symmetric encryption initialization vector pair. This pair is distinct for each chunk. A encrypts the random key with K_{SA}, as $e = Enc_{K_{SA}}(k_{\langle c \rangle})$. Next, A hashes the ciphertext C as $H(C)$. Subsequently, it computes its commitment to the encrypted chunk and the encrypted key as $T_{AS} = MAC_{K_{SA}}[\langle A \rangle, \langle B \rangle, \langle F \rangle, \langle c \rangle, e, H(C), t]$. The tickets T_{AS} are only valid for a certain amount of time T_{key}, which forces B to expedite paying for decrypting the encrypted chunks. This fact allows A to promptly acquire credit for its service. A sends the following to B.

$A \longrightarrow B$: [chunk response]$T_{AS}, \langle F \rangle, \langle c \rangle, e, C, t, p_A$

Step 7: To retrieve $k_{\langle c \rangle}$, B needs to request it from the server. As soon as B receives the encrypted chunk, B computes its own hash over the received ciphertext C' and forwards the following to S.

$B \longrightarrow S$:[decryption key request] $\langle A \rangle, \langle F \rangle, \langle c \rangle, e, H(C'), t, T_{AS}, p_A$

Step 8: If *current-time* $\leq t + T_{key}$, and p_A is not too much off, S checks if $T_{AS} = MAC_{K_{SA}}[\langle A \rangle, \langle B \rangle, \langle F \rangle, \langle c \rangle, e, H(C'), t]$. 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 punishes neither AS or B 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 chunk response 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 content $\langle F \rangle$. If B is approved, it charges B and rewards A with Δ_c credit units. Subsequently, S decrypts $k'_{\langle c \rangle} = Dec_{K_{SA}}(e)$, and sends it to B.

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

B uses $k'_{\langle c \rangle}$ to decrypt the chunk as $c' = Dec_{k'_{\langle c \rangle}}(C')$. Next, we explain the complaint mechanism.

Step 9: If the decryption fails or if $H(c') \neq H(c)$ (step 2), B complains to S by sending the following message.

$B \longrightarrow S$:[complaint] $\langle A \rangle, \langle F \rangle, \langle c \rangle, T_{AS}, e, H(C'), t, p_A$

S ignores this message if *current-time* $> t + T'_{key}$, where $T'_{key} > T_{key}$. $T'_{key} - T_{key}$ 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. With this condition, a misbehaving client 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_{key}. 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* $> t + T'$.

If $T_{AS} \neq MAC_{K_{SA}}[\langle A \rangle, \langle B \rangle, \langle F \rangle, \langle c \rangle, e, H(C'), t]$, 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 $k'_{\langle c \rangle} = Dec_{K_{SA}}(e)$ once again, retrieves c from its storage, and encrypts c himself using $k'_{\langle c \rangle}$, $C'' = Enc_{k'_{\langle c \rangle}}(c)$. 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 tickets verify, 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.

3.5 Other Protocol Issues

A content provider may be more concerned with scalability than it is with the free-riding problem presented in Section 2. In such case, it can deploy clients that use tit-for-tat incentives if their peers have content that interests them, i.e. the clients would upload plaintext content to peers that reciprocate with plaintext content. These clients would fall back to Dandelion's cryptographic fair-exchange mechanism when their peers do not have content that interests them. For example, selfish seeders would always upload encrypted content to their peers.

In case a client is unable to timely retrieve a missing chunk from its peers, it resorts to requesting the chunk from the server. If the server is not busy, it replies with the plaintext chunk. If it is moderately busy, it charges an appropriately large amount of credit $\Delta_s > \Delta_c$, sends the chunk and indicates that it is preferable for the client not to download chunks from the server. If the server is overloaded, it replies with an error message. Clients always download the content from the server in chunks, so that the system can seamlessly switch to the peer-serving mode when the server becomes busy.

Typically, a content distributor would deploy, in addition to the server, at least one client that possesses the complete content (initial seeder). In this way, the distributor ensures that the complete content is made available, even if the server is too busy to serve chunks.

4 Security Analysis

This section briefly lists the security properties of our design. For simplicity of presentation, we omit proofs on why these properties hold. They can be found in [50].

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.

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 Δ_c credit units from S.

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.

Lemma 4.4 A malicious client cannot replay previously sent valid requests to the server or generate decryption key 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.

Observation 4.5 A client cannot download chunks from a selfish peer if it does not have sufficient credit. Our design choice to involve the server in every transaction, instead of using the fair exchange technique proposed in [42], enables the server to check a client's credit balance before the client retrieves the decryption key of 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 error or due to client misbehavior. The performance of the system would be severely degraded if clients 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 B can always abandon any instance of the protocol. In such case, A does not receive any credit, as argued in Lemmas 4.1 to 4.3, even though B may have consumed A's resources. This is a denial of service attack against A. Note that this attack would require that the malicious client B expends resources proportional to the resources of the victim A, 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 a short-lived ticket T_{SA} to authorized clients only. T_{SA} is checked for validity by A (steps 4 and 6 in Section 3.4.2). In addition, S may charge an authorized B for the issuance of tickets T_{SA} effectively deterring B 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.

5 Implementation

This section describes a prototype C implementation of Dandelion that is suitable for cooperative content distribution of static content. It uses the *openssl* [5] library for cryptographic operations and standard file I/O system calls to efficiently manage credit information, which is stored in a simple file.

5.1 Server Implementation

The server and the credit base are logical modules and could be distributed over a cluster to improve scalability. For simplicity, our current implementation combines the content provider and the credit base at a single server.

The server implementation is single-threaded and event-driven. The network operations are asynchronous, and data are transmitted over TCP. In order to scale to thousands of simultaneously connected clients, the server employs the *epoll()* event mechanism.

Each client is assigned an entry in a credit file, which stores the credit as well as authentication and file access control information. Each entry has the same size and the client ID determines the offset of the entry of each client in the file, thus each entry can be efficiently accessed for both queries and updates.

The server queries and updates a client's credit from and to the credit file upon every transaction, Yet, it does not force commitment of the update to persistent storage. Instead, it relies on the OS to perform the commitment. If the server application crashes, the update will still be copied from the kernel buffer to persistent storage. Still, the OS may crash or the server may lose power before the updated data have been committed. However, in practice, a typical Dandelion deployment would run a stable operating system and use backup power supply. In addition, the server could mirror the credit base on multiple machines using high speed IP/Ethernet I/O. In addition, transactions would not involve very large amounts

of money per user. Hence, we believe it is preferable not to incur the high cost of committing the credit updates to non-volatile memory after every transaction or after a few transactions (operations 12 and 13 in Table 1).

5.2 Client Implementation

The client side is also single-threaded and event-driven. A client may leech or seed multiple files at a time. A client can be decomposed into two logical modules: a) the *connection management* module; and b) the *peer-serving* module.

The connection management module performs *peering* and *uploader discovery*. With peering, each client obtains a random partial swarm view from the server and strives to connect to $O(\log n)$ peers, where n is the number of nodes in the Dandelion swarm, as communicated to the node by the server. As a result, the swarm approximates a random graph with logarithmic out-degree, which has been shown to have high connectivity [21]. With uploader discovery, a client attempts to remain connected to a minimum number of uploading peers. If the number of recent uploaders drops below a threshold, a client requests from the server a new swarm view and connects to the peers in the new view.

The peer-serving module performs *content reconciliation* and *downloader selection*. Content reconciliation refers to the client functionality for announcing recently received chunks, requesting missing chunks, requesting decryption keys for received encrypted chunks, and replying to chunk requests. Our implementation employs locally-rarest-random [39] scheduling in requesting missing chunks from clients. To efficiently utilize their downlink bandwidth using TCP, clients strive to at all times keep a specified number of outstanding chunk requests [26, 39], which have been sent to a peer and have not been responded to.

With downloader selection, the system aims at making chunks available to the network as soon as possible. In the following description, n denotes the number of parallel downloaders. Dandelion's downloader selection algorithm is similar to the *seeder* choking algorithm used in the "vanilla" BitTorrent 4.0.2, as documented in [40]. The algorithm is executed by each client every 10 seconds. It is also executed when a when a peer that is selected to be downloader disconnects. The algorithm proceeds as follows: a) peers that are interested in the client's content are ranked based on the time they were last selected to be downloaders (most recent first); b) the client selects as downloaders the $n-2$ top ranked peers; c) in case of a tie, the peer with the highest download rate from the client is ranked higher; and d) the client randomly selects two additional downloaders from the non-selected nodes that are interested in the client's content. Step (d) is performed in expectation of discovering fast downloaders and to jumpstart peers that recently joined the swarm.

This downloader selection algorithm aims at reducing the amount of duplicate data a client needs to upload, before it has uploaded a full copy of its content to the swarm. Downloader selection improves the system's performance in the following additional ways. First, it limits the number of peers a client concurrently uploads to, such that complete chunks are made available to other peers and relayed by them at faster rates. Second, given that all clients are connected to roughly the same number of peers, it also limits the number of peers a client concurrently downloads from to approximately n. As a result, the rate with which the client downloads complete chunks increases. Last, by limiting the number of connections over which clients upload, it avoids the inefficiency and unfairness that is observed when many TCP flows share a bottleneck link [45].

The number of peers that download from a client in parallel depends on the client's upload bandwidth. We have empirically determined that a good value for this parameter in the PlanetLab environment is 10.

6 Evaluation

The goal of our experimental evaluation is to demonstrate the viability and to identify the scalability limits of Dandelion's centralized and non-manipulable virtual-currency-based incentives.

6.1 Dandelion Profiling

We profile the cost of operations performed by the server and clients aiming at identifying the performance bottlenecks of our design. The measurements are performed on a dual Pentium D 2.8GHZ/1MB CPU with 1GB RAM and 250GB/7200RPM HDD running Linux 2.6.5-1.358smp.

Table 1 lists the cost of per chunk Dandelion operations. In a flash crowd event, the main task of a Dandelion server is to: a) receive the decryption key request (operation 7); b) authenticate the request by computing an HMAC (operation 1); c) verify the ticket by computing an HMAC (operation 2); d) decrypt the encrypted decryption key (operation 3); e) query and update the credit of the two clients involved (operations 10 and 11); f) sign the decryption key response (operation 4); and g) send the decryption key response (operation 8).

As can be seen in the table, the per chunk cryptographic operations of the server (operations 1-4) are highly efficient (total 109 usec), as only symmetric cryptography is employed. The credit management operations (10 and 11) are also efficient (total 24 usec). On the other hand, the communication costs clearly dominate the processing costs, indicating that for 1Mb/s uplink and downlink, the downlink is the bottleneck.

	Dandelion operation	Size	Time (ms)
	CPU-centric Operation		
1	Authenticate decryption key request	98 bytes	.018
2	Generate ticket for decryption key key request or complaint verification	78 bytes	.018
3	Encrypt/decrypt decryption key	32 bytes	.056
4	Sign decryption key response	46 bytes	.017
5	Encrypt/decrypt chunk	128 KB	.715
6	Hash encrypted chunk for commitment generation or for commitment processing	128 KB	.487
	Communication Operation		
7	Receive decryption key request	156 bytes	∼1.79
8	Transmit decryption key response	104 bytes	∼1.39
9	Transmit chunk	128 KB	∼1053
	Credit Management Operation		
10	Query credit file	N/A	∼0.004
11	Update credit file without commit to disk (rely on OS)	N/A	∼0.02
12	Update credit file and commit to disk	N/A	∼9.25
13	Update credit file and commit to disk every 100 updates	N/A	∼0.27

Table 1: **Timings of per chunk transaction Dandelion operations. Timings for operations 1-6 are obtained using *getrusage(RUSAGE_SELF)* over 10000 executions to obtain 1 usec precision. Timings for operations 7-9 are approximated according to our application layer rate-limiting for 1Mb/s uplink and 1Mb/s downlink. They are provided as reference for comparison with CPU-centric and credit management operations. Timings for operations 10-13 are approximated using *gettimeofday()* over 10000 executions. Operations 3 and 5 use 8-byte-block Blowfish-CBC with 128-bit key and 128-bit initialization vector.1, 2 and 4 use HMAC-SHA1 with 128-bit key. Operation 6 uses SHA-1. Operations 10-12 are performed on a credit file with 10000 44-byte entries. For committing to disk in operations 12 and 13, we use fsync() and we disable HDD caching.**

The cost of a complaint is higher because in addition to verifying a ticket, it involves reading a chunk, encrypting it with the sender client's key (operation 5), and hashing the encrypted chunk (operation 6).

A Dandelion client can decrypt/encrypt and hash a 128KB chunk (operations 5 and 6) much faster than download it or transmit it at 1Mb/s (operation 9). Therefore, the client's processing overhead does not affect its upload or download throughput.

Note that this profiling repeats the same operation multiple times. It does not consider the parallel processing of I/O and CPU operations. In addition, it does not include the cost of system calls and the cost of TCP/IP stack processing. Therefore, we refrain from deriving conclusions on the throughput of the server. Such conclusions are derived in the subsequent evaluation.

6.2 Server Performance

A Dandelion server mediates the chunk exchanges between its clients. The download throughput of clients in our system is bound by how fast a server can process their decryption key requests. Both the server's computational resources and bandwidth may become the performance bottleneck.

We deploy a Dandelion server that runs on the same machine as the one used for Dandelion profiling. We also deploy ∼200 clients that run on distinct PlanetLab hosts. The server machine shares a 100Mb/s Ethernet II link. To mitigate bandwidth variability in the shared link and to emulate a low cost server with an uplink and downlink that ranges from 1Mb/s to 5Mb/s, we rate-limit the server at the application layer.

In each experiment, the clients send requests for decryption keys to the server and we measure the aggregate rate with which all clients receive key responses. The server always queries and updates the credit base from and to the credit file without forcing commitment to disk. The specified per client request rate varies from 1 to 14 requests per second. For each request rate, the experiment duration was 10 minutes and the results were averaged over 10 runs. As the request rate increases and the server's receiver buffers become full, clients do not send new requests at the specified rate, due to TCP's flow control. When the request rate increases to the point that the server's resources become saturated, the key response rate from the server decreases.

Figure 3(a) depicts the server's decryption key throughput for various server bandwidth capacities. As the bandwidth increases from 1Mb/s to 3Mb/s, the server's decryption key response throughput increases. This indicates that for 1Mb/s to 3Mb/s access links, the bottleneck is the bandwidth. When the bandwidth limit is 4Mb/s and 5Mb/s, the server exhibits similar performance, which suggests that the access link is not saturated at 4Mb/s. The results show that a server running on our commodity PC with 4Mb/s or 5Mb/s access link can process up to ∼1200 decryption key requests per second. This indicates that with a 128KB chunk size, this server may simultaneously support almost 1200 clients that download from each other at 128KB/s. With a larger chunk size, each such client sends decryption key requests at a slower rate, and the number of supported clients increases.

Figures 3(b) and 3(c) show the average CPU and memory utilization at the server over the duration of the above experiments. We observe that for 4Mb/s and 5Mb/s, the server's CPU utilization reaches ∼100%, indicating that the bottleneck is the CPU. In Figure 3(c), we see that the server consumes a very small portion of the available memory.

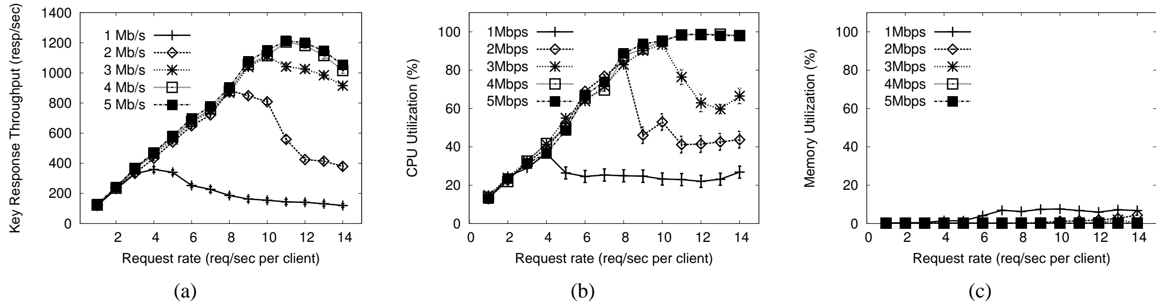

Figure 3: **(a) Aggregate decryption key response throughput of the Dandelion server as a function of per-client key request rate, for varying server rate-limits; (b) server's CPU utilization(%) as a function of per-client decryption key request rate; (c) server's memory utilization (%) as a function of per-client decryption key request rate.**

6.3 System Performance

The following experiments evaluate the performance of the Dandelion system on PlanetLab. We examine the impact of chunk size and the impact of seeding on the performance of the system. We also compare our system's performance to BitTorrent's. In all experiments we ran a Dandelion server within a PlanetLab VServer spawned on a highly available Xeon 3GHZ/2MB CPU and 2GB RAM machine. We rate-limit the server at 2Mb/s.

Leechers are given sufficient initial credit to completely download a file. Clients always respond to chunk requests from their selected downloaders and they never request chunks from the server. We do not rate-limit the Dandelion and BitTorrent clients, as a means for testing our system in heterogeneous Internet environments. To cover the bandwidth-delay product in PlanetLab, the TCP sender and receiver buffer size is set equal to 120KB.

For each configuration we repeat the experiment 10 times and we extract mean values and 95% confidence intervals over the swarm-wide mean file download completion times of each run. The file download completion time is the time that elapses between the moment the client contacts the server to start a download and the moment its download is completed.

6.3.1 Selecting Chunk Size

This experiment aims at examining the tradeoffs involved in selecting the size of the chunk, the verifiable transaction unit in Dandelion. Intuitively, since clients are able to serve a chunk only as soon as they complete its download, a smaller chunk size yields a more efficient distribution pipeline. In addition, when the file is divided into many pieces, chunk scheduling techniques such as rarest-first can be more effective, as there is sufficient content entropy in the network. Consequently, clients can promptly discover and download content of interest. However, a smaller chunk size increases the rate with which key requests are sent to the server, reducing the scalability of the system. In addition, due to TCP's slow start, a small chunk size cannot ensure high bandwidth utilization during the TCP transfer of any chunk.

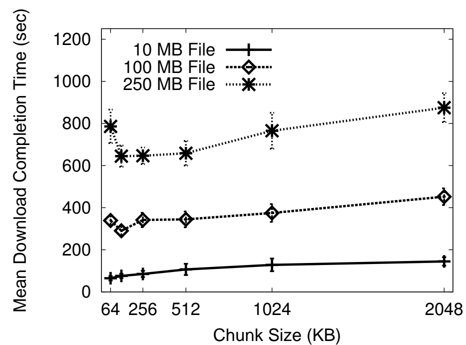

Figure 4: **Mean file download completion times of 40 leechers as a function of chunk size. The swarm has one initial seeder. The *x* axis corresponds to the chunk size. The *y* axis corresponds to the mean download completion time in the swarm. The error bars correspond to 95% confidence intervals.**

In each configuration, we deploy approximately 40 Dandelion leechers and one initial seeder. Leechers start downloading files almost simultaneously. We deploy only 40 leechers to ensure that the server is not saturated, even if we use 64KB chunk size.

Figure 4 shows the leecher mean download completion time as a function of the chunk size. For smaller files, e.g., the 10MB file, the system has the best performance for chunk size equal to 64KB. The system's performance degrades with the chunk size. As the file size increases, the beneficial impact of small chunks, becomes less significant. For example, for 250MB file, a 128KB chunk size yields notably better performance than a 64KB chunk size.

Based on the above results and further fine-tuning, in the rest of this evaluation, we use 128KB chunks.

6.3.2 Impact of Dandelion Seeders

One of Dandelion's main advantages is that it provides robust incentives for clients to seed. We quantify the performance gains from the existence of seeders in our system. In each experiment, we deploy ∼200 leechers. Leechers start downloading the file almost simultaneously, creating a flash crowd.

We show the impact of seeders by varying the probability that a leecher remains online to seed a file after it completes its download. In each experiment, a swarm has one initial seeder. Upon completion of its download, each leecher stays in the swarm and seeds with probability a. Probability a varies in 25% and 100%.

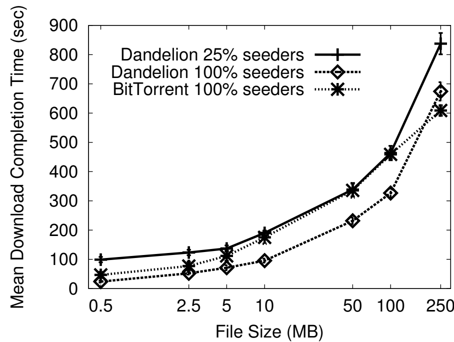

Figure 5: **Swarm-wide mean file download completion times of ∼200 leechers as a function of file size for varying portion of leechers that become seeders. The error bars correspond to 95% confidence intervals.**

Figure 5 depicts the mean download completion time over all ∼200 leechers as a function of the file size, for varying a. The results show the beneficial impact of seeders and the importance of a mechanism to robustly incent seeding. For example, for a 250MB file, we observe a swarm-wide mean download completion time of 674 sec and 837 sec when leechers become seeders with 100% and 25% probability, respectively. If we express the impact of seeders as the ratio of the mean download time for $a = 100\%$ over the mean download time for $a = 25\%$, we observe that the impact is reduced as the file size increases. The larger the file is, the longer clients remain online to download it, resulting in clients contributing their upload bandwidth for longer periods. For smaller files however, leechers rely heavily on the initial seeder and the leechers that become seeders to download their content from. Therefore for small files, a reduction in probability a results in substantially longer download completion times.

6.3.3 Comparison with BitTorrent

Figure 5 also shows the download completion times of ∼200 tit-for-tat compliant CTorrent 1.3.4 leechers. All BitTorrent leechers remain online after their download completion ($a = 100\%$). The purpose of this illustration is to show that Dandelion can attain performance comparable to the one achieved by BitTorrent, although it employs a different downloader selection algorithm and involves the server in each chunk exchange.

Although Dandelion appears to outperform BitTorrent for certain file sizes, we do not claim that it is in general a better-performing protocol. The performance of both protocols is highly dependent on numerous parameters, which we have not exhaustively analyzed.

6.3.4 Credit Distribution

We examine the distribution of credit during a Dandelion file distribution. The purpose of this measurement is to identify which types of clients tend to accumulate the most credit in swarms of similar configuration to ours.

Figure 6 shows the scatter plot of the client's credit after a single 250MB file download by ∼200 leechers together with the mean download rate of each client. In the experiment, there is only one initial seeder. All nodes are given 100% of the credit needed to download the file and they all become seeders upon download completion. We observe that the seeder obtained the most credit during the file distribution. This is expected, as a seeder is always in position to upload useful content to its peers and our seeder had a fast access link. Since fast downloaders obtain useful content earlier in the distribution and are likely to have uplinks proportional to their downlink, they should be able to deliver more content and earn more credit. Our results confirm this intuition and show that there is *strong* correlation between average download rate and credit ratio, i.e. the product-moment correlation coefficient is equal to 0.72.

We also observe that many clients uploaded substantially less than they downloaded. Indicatively, 25.8% of the clients had less than 70% of their initial credit.

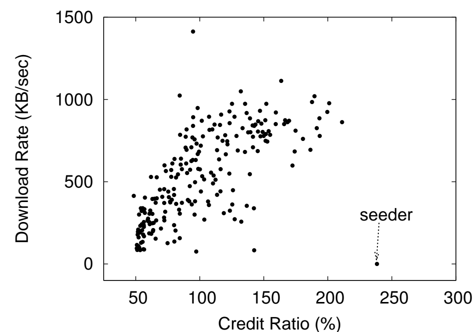

Figure 6: **The scatter plot of the distribution of credit after ∼200 leechers have completed a 250MB file download and their average download rates. The *x* axis shows the credit ratio, which is the ratio of the remaining credit of a client over its initial credit. The *y* axis shows the average download rate of each client. The seeder is included for illustration purposes, but its download rate is invalid.**

7 Conclusion

This paper describes Dandelion: a cooperative (P2P) system for the distribution of paid content. Dandelion's primary function is to enable a content provider to provide strong incentives for clients to contribute their uplink bandwidth.

Dandelion rewards selfish clients with virtual currency (credit) when they upload valid content to their peers and charges clients when they download content from selfish peers. Since Dandelion employs a non-manipulable cryptographic scheme for the fair exchange of content uploads for credit, the content provider is able to redeem a client's credit for monetary rewards. Thus, it provides strong incentives for clients to seed content and to not free-ride.

Our experimental results show that seeding substantially improves swarm-wide performance. They also show that a Dandelion server running on commodity hardware and with moderate bandwidth can scale to a few thousand clients. Dandelion's deployment in medium size swarms demonstrates that it can attain performance that is comparable to BitTorrent. These facts demonstrate the plausibility of our design choice: centralizing the incentive mechanism in order to increase resource availability in P2P content distribution networks.

8 Acknowledgements

We are thankful to Nikitas Liogkas, Eddie Kohler and the anonymous reviewers for their extensive and fruitful feedback. We also thank Rex Chen, Dehn Sy and Lichun Bao with Calit2 for providing space and equipment for our experiments. This work was supported in part by NSF award CNS-0627166.

References

[1] http://www.ietf.org/rfc/rfc4346.txt.

[2] Enhanced CTorrent. www.rahul.net/dholmes/ctorrent/.

[3] Kazaa. www.kazaa.com.

[4] Kazaa Lite. en.wikipedia.org/wiki/Kazaa_Lite.

[5] Openssl. www.openssl.org.

[6] Peer exchange. www.azureuswiki.com/index.php/Peer_Exchange.

[7] The eMule Project. www.emule-project.net.

[8] The MNet Project. mnetproject.org.

[9] Trackerless BitTorrent. www.bittorrent.com/trackerless.html.

[10] iTunes outsells CD stores as digital revolution gathers pace. arts.guardian.co.uk/netmusic/story/0,,1649421,00.html, 2005.

[11] BitTorrent Strikes Digital Download Deals with 20th Century Fox, G4, Kadokawa, Lionsgate, MTV Networks, Palm Pictures, Paramount and Starz Media. home.businesswire.com/portal/site/google/index.jsp?ndmViewId=news_view&newsId=20061128006262&newsLang=en, 2006.

[12] EMI Music makes its catalog available to Qtrax: the world's first ad-supported, legitimate P2P service. www.emigroup.com/Press/2006/press25.htm, June 2006.

[13] Music denied - Shoppers overwhelm iTunes. edition.cnn.com/2006/TECH/internet/12/28/itunes.slowdown.ap/index.html?eref=rss_topstories, 2006.

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

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

[16] Warner Bros to sell films via BitTorrent. www.msnbc.msn.com/id/12694081/, June 2006.

[17] N. Andrade, M. Mowbray, A. Lima, G. Wagner, and M. Ripeanu. Influences on Cooperation in Bittorrent Communities. In *P2P Econ*, 2005.

[18] N. Asokan, M. Schunter, and M. Waidner. Optimistic Protocols for Fair Exchange. In *CCS*, 1997.

[19] F. Bao, R. Deng, and W. Mao. Efficient and Practical Fair Exchange Protocols with Off-line TTP. In *S&P*, 1998.

[20] M. Bellare, R. Canetti, and H. Krawczyk. Keying Hash Functions for Message Authentication. In *LNCS*, volume 1109, 1996.

[21] B. Bollobas. Random Graphs. In *Academic Press*, 1985.

[22] E. F. Brickell, D. Chaum, I. Damg, and J. V. de Graaf. Gradual and Verifiable Release of a Secret. In *CRYPTO*, 1988.

[23] B. Cheng, X. Liu, Z. Zhang, and H. Jin. A measurement study of a peer-to-peer video-on-demand system. In *IPTPS*, 2007.

[24] 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*, 2003.

[25] R. Cleve. Controlled Gradual Disclosure schemes for Random bits and their Applications. In *CRYPTO*, 1989.

[26] B. Cohen. Incentives Build Robustness in BitTorrent. In *P2P Econ*, 2003.

[27] I. B. Damg. Practical and Provably Secure Release of a Secret and Exchange of Signatures. In *EUROCRYPT*, 1994.

[28] J. R. Douceur. The Sybil Attack. In *IPTPS*, 2002.

[29] P. Druschel, A. Nandi, T.-W. J. Ngan, A. Singh, and D. Wallach. Scrivener: Providing Incentives in Cooperative Content Distribution Systems. In *Middleware*, 2005.

[30] S. Even, O. Goldreich, and A. Lempel. A Randomized Protocol for Signing Contracts. In *Communications of the ACM*, 1985.

[31] B. Fan, D.-M. Chiu, and J. C. Lui. The Delicate Tradeoffs in BitTorrent-like File Sharing Protocol Design. In *ICNP*, 2006.

[32] K. Franklin and M. K. Reiter. Fair exchange with a semi-trusted third party. In *CCS*, 1997.

[33] M. K. Franklin and G. Tsudik. Secure Group Barter: Multi-party Fair Exchange with Semi-Trusted Neutral Parties. In *FC'98*, 1998.

[34] J. Gray. Distributed Computing Economics. Technical report, Microsoft Research, 2003. MSR-TR-2003-24.

[35] B. Horne, B. Pinkas, and T. Sander. Escrow Services and Incentives in Peer-to-peer networks. In *EC*, 2001.

[36] D. Hughes, G. Coulson, and J. Walkerdine. Free Riding on Gnutella Revisited: The Bell Tolls? In *IEEE Distributed Systems Online*, 2005.

[37] S. Jun and M. Ahamad. Incentives in BitTorrent Induce Free Riding. In *P2P Econ*, 2005.

[38] I. Keidar, R. Melamed, and A. Orda. EquiCast: Scalable Multicast with Selfish Users. In *PODC*, 2006.

[39] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Maintaining High Bandwidth Under Dynamic Network Conditions. In *USENIX*, 2005.

[40] A. Legout, N. Liogkas, E. Kohler, and L. Zhang. Clustering and Sharing Incentives in BitTorrent Systems. SIGMETRICS, 2007.

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

[42] J. Li and X. Kang. Proof of Service in a Hybrid P2P Environment. In *ISPA Workshops*, 2005.

[43] N. Liogkas, R. Nelson, E. Kohler, and L. Zhang. Exploiting BitTorrent For Fun (But Not Profit). In *IPTPS*, 2006.

[44] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free Riding in BitTorrent is Cheap. In *HotNets*, November 2006.

[45] R. Morris. TCP Behavior with Many Flows. In *ICNP*, 1997.

[46] T. Okamoto and K. Ohta. How to Simultaneously Exchange Secrets by General Assumptions. In *CCS*, 1994.

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

[48] J. Shneidman, D. Parkes, and L. Massoulie. Faithfulness in Internet Algorithms. In *PINS*, 2004.

[49] M. Sirivianos, J. H. Park, R. Chen, and X. Yang. Free-riding in BitTorrent Networks with the Large View Exploit. In *IPTPS, www.ics.uci.edu/~msirivia/publications/large-view.pdf*, 2007.

[50] M. Sirivianos, J. H. Park, X. Yang, and S. Jarecki. Dandelion: Cooperative Content Distribution with Robust Incentives. UCI-ICS TR 07-06, www.ics.uci.edu/~msirivia/publications/dandelion-usenix-tr.pdf, 2007.

[51] K. Tamilmani, V. Pai, and A. Mohr. SWIFT: A System with Incentives for Trading. In *P2P Econ*, 2004.

[52] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer. KARMA: A Secure Economic Framework for P2P Resource Sharing. In *P2P Econ*, 2003.

[53] K. Wei, Y.-F. Chen, A. J. Smith, and B. Vo. WhoPay: a Scalable and Anonymous Payment System for Peer-to-Peer Environments. In *ICDCS*, 2006.

[54] B. Yang and H. Garcia-Molina. PPay: Micropayments for Peer-to-peer Systems. In *CCS*, 2003.

[55] J. Zhou and D. Gollmann. A Fair Non-repudiation Protocol. In *S&P*, 1996.

[56] J. Zhou and D. Gollmann. An Efficient Non-repudiation Protocol. In *CSFW*, 1996.