Robust and Efficient Incentives for Cooperative Content Distribution

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

***Abstract*—Content distribution via the Internet is becoming increasingly popular. To be cost-effective, commercial content providers are now using peer-to-peer (P2P) protocols such as BitTorrent to save bandwidth costs and to handle peak demands. When an online content provider uses a P2P protocol, it faces an incentive issue: how to motivate its clients to upload to their peers.**

This paper presents Dandelion, a system designed to address this issue. Unlike previous incentive-compatible systems, such as BitTorrent, our system provides non-manipulable incentives for clients to upload to their peers. A client that honestly uploads to its peers is rewarded in the following two ways. First, if its peers are unable to reciprocate its uploads, the content provider rewards the client's service with credit. This credit can be redeemed for discounts on paid content or other monetary rewards. Second, if the client's peers possess content of interest and have appropriate uplink capacity, the client is rewarded with reciprocal uploads from its peers.

In designing Dandelion, we trade scalability for the ability to provide robust incentives for cooperation. 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. The download completion time for these clients is substantially reduced due to the additional upload capacity offered by strongly incentivized uploaders.

***Index Terms*—Peer-to-peer, content distribution, incentives, fair-exchange, symmetric cryptography.**

I. Introduction

Content distribution via the Internet is becoming increasingly popular among the entertainment industry and the consumers alike. For example, Hulu [1] streams authorized content for NBC, Fox and other networks. However, the increasing demand for digital content is overwhelming the infrastructure of online content providers [2]. 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 over-provision its bandwidth to handle peak demands, nor does it require the provider to rely solely on purchased service from a third-party such as Akamai. Instead, a P2P protocol such as BitTorrent [3] harnesses its clients' unused uplink bandwidth, and saves the bandwidth and computing resources of a content provider. Huang et al. [4] showed that peer-assisted content distribution can substantially reduce the operating costs of Video on Demand services. To that effect, BBC has successfully launched its iPlayer peer-assisted VoD service, and leading content providers have now partnered with BitTorrent, Inc [5]. This trend indicates that P2P protocols enable a site to cost-effectively distribute content.

When an online content provider uses a P2P protocol, it faces an incentive 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. In addition, in a competitive market, a content provider with paying customers needs to offer better quality of service guarantees than the ones offered by free P2P content distribution systems. However, selfish (rational) users tend not to share their bandwidth without external incentives [6]. Although the popular BitTorrent protocol, has incorporated the rate-based tit-for-tat (TFT) incentive mechanism, this mechanism bears two weaknesses. First and foremost, 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 [7–10], allowing modified clients to free-ride and still achieve download rates equal to or higher than the ones of cooperative clients (§ II-B,VI-C).

In previous work, we introduced Dandelion [11], a protocol that provides provably non-manipulable incentives for seeding and is not susceptible to free-riding. Although the protocol was shown to be sufficiently scalable, its incentive mechanism was completely centralized. In this paper, we built upon our initial design and propose changes that *partially* decentralize the protocol. These changes render Dandelion more scalable, while they maintain its original desirable properties: robust incentives for cooperation and performance comparable to the most efficient content distribution systems to date. Our modified protocol provides robust incentives using two mechanisms.

The first mechanism guarantees strict fair-exchange of content uploads for real monetary value. This mechanism is useful when a client has content that interests its peer but the peer has no content of interest to reciprocate with. *Selfish* clients (i.e. rational clients that do not upload unless they expect to be rewarded) earn credit when they upload valid content to their peers. Credit can be redeemed at a content provider for discounts on the content or for other types of monetary awards. Given appropriate pricing schemes, we expect that a selfish client is motivated to serve content to its peers. We refer to this mechanism as *credit-based exchange*.

The second mechanism renders the protocol more scalable by partially decentralizing it, while preventing free-riding. It

enables clients that are mutually interested in each other's content to barter their uplink bandwidth. We refer to the second mechanism as *Tit-for-tat-based (TFT-based) exchange*.

A key challenge lies in making the exchange of content uploads for credit efficient and practical, while robust to manipulation. Practice has shown that this problem is a major stumbling block for the commercial adoption of micropayment-based incentive schemes [12]. Manipulability is a primary contributor to the weak adoption of credit-based (micropayment-based) incentive schemes for commercial purposes because it makes content distributors wary of substantial monetary losses in case the client software is compromised.

We address this challenge based on the insight that the content provider itself is a trusted third party (TTP) and can mediate the content exchange between its clients. Under the credit-based exchange protocol, clients exchange data for credit and a server mediates the transaction. The server uses only efficient symmetric cryptography on critical data paths and sends only short messages to its clients.

In our setting, any unfairness during TFT exchanges results in monetary losses for the peer that does not receive its deserved reciprocation. We address this issue using an optimistic fair-exchange protocol that is an adaptation of BAR Gossip [13] and classic optimistic fair-exchange exchange schemes [14, 15]. In optimistic fair-exchange, the trusted third party is involved only when an error occurs or when dishonest participants do not follow the protocol. Implementing a simple scheme chunk-level tit-for-tat exchanges of plaintext content would not guarantee absolute fairness, since the last peer that receives content may refrain from reciprocating.

As a side-effect of the server-mediated fair-exchange, Dandelion discourages unauthorized content distribution in the sense that it gives no incentives for seeders to waste their bandwidth for uploads to unauthorized clients. Instead, it provides explicit rewards for them to upload to authorized clients.

Our work makes the following contributions:
1) The design of Dandelion, a hybrid incentive scheme for commercial P2P content distribution. It combines an efficient fair-exchange scheme that enables trading of content uploads for credit with an optimistic fair-exchange scheme that enables the bartering of uplink bandwidth.
2) The prototype implementation of a Dandelion-based system that is suitable for P2P distribution of static content.
3) The evaluation of our implementation on PlanetLab [16], which identifies the scalability limits of our incentive mechanism and demonstrates the plausibility of our approach. In addition, it thoroughly investigates free-riding in realistic PlanetLab-residing BitTorrent-like networks.

The rest of this paper is organized as follows. Section II provides background and motivates our design. Section III provides an overview of Dandelion and describes the system model under which it is designed to operate. Section IV describes the design of Dandelion and discusses its properties. Section V describes the implementation of our prototype system. Section VI presents the experimental evaluation of our implementation. In Section VII we discuss prior work and we conclude in Section VIII.

II. Background

Dandelion's design addresses the incentive issues in P2P content distribution protocols such as BitTorrent [3] and eMule [17]. In this section, we motivate the design of Dandelion by discussing the weaknesses of BitTorrent's incentive mechanism.

In the rest of this paper we use a BitTorrent-like terminology. A *seeder* refers to a client that uploads to its peers after it has completed its download. A *leecher* is a client that has not completed its download. A *free-rider* refers to a client that downloads content from other peers without incurring any cost, i.e without uploading content or without expending currency. A *swarm* refers to all clients that actively participate in the protocol for a given content item. The *choking algorithm* refers to the client-side function of selecting peers to upload content to (unchoke) in parallel, based on a predetermined criterion. *Optimistic unchoking* refers to temporarily unchoking a peer, although that peer does not currently satisfy the unchoking criterion. Clients optimistically unchoke a peer in expectation that the peer will eventually satisfy the unchoking criterion.

A. Impact of Seeding

The popular BitTorrent protocol employs the rate-based "tit-for-tat" (TFT) incentive mechanism [3]. A peer prefers to upload to (unchoke) another peer that reciprocally uploads parts of the same file. 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 [18], and use this history to decide which clients can join a swarm, practice has shown that clients can fake their upload history [19, 20] or collude [21].

Seeders improve download completion times, because they increase the content availability and the aggregate upload bandwidth. In addition, incentives for seeding are crucial because a large portion of P2P clients in the Internet reside behind asymmetric links. This means that the total upload capacity of the P2P network may be much lower than its total download capacity. However, lack of incentives leads to BitTorrent swarms being underprovisioned in terms of seeders [22]. In order to rectify this situation one needs to persuade peers to remain online to seed after they complete their download. In § VI-B2, we show that the download rates of leechers in BitTorrent swarms increases substantially as the number of clients that seed increases.

These observations are corroborated by recent and older measurements in P2P content distribution systems. Izal et al. [23] analyzed the lifetime of a healthy torrent with many seeders and derived that almost 40% of the file was uploaded by seeders and that on average it took clients twice as much time to upload than to download the same quantity of bytes. A recent measurement [24] over 1000 torrents and 100000 peers revealed that altruistic nodes that uploaded twice as much as they downloaded by remaining as seeders comprised only 17% of their swarm. Piatek et al. [22] showed that clients that joined 13353 swarms and contributed 100KB/sec

achieved a median download rate of only 14KB/sec and in 25% of the swarms they were not able to download at all. In addition, they found that 20% of 55,523 swarms had less than 1 seeder per 10 peers. They conclude that the vast majority of swarms would have significantly more availability and upload capacity if downloaders were incentivized to upload after download completion. Furthermore, experience with the Maze system [25] has shown that demand for content is heavily long-tailed; around 80% of downloading involves at most one downloader, therefore it is crucial to keep seeders online.

B. Free-riding in BitTorrent

A general observation is that since BitTorrent's tit-for-tat incentives reward cooperative leechers with improved download times, leechers are always incentivized to upload. This observation relies on the assumption that users aim only at maximizing their download rates. However in practice, BitTorrent users may 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 more critical tasks.

Considering the trade-off between performance and susceptibility to free-riding [26], BitTorrent purposely does not implement a strict 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 (30 seconds). Furthermore, 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 [7, 8], we employ the "large view" exploit [9] to free-ride in BitTorrent-like swarms. The free-rider client obtains a larger than normal view of the network and connects to all peers in its view, while it does not upload any content. Using this exploit in a sufficiently large swarm, a free-rider can find more seeders, which do not employ tit-for-tat. It can also increase the frequency with which it becomes optimistically unchoked, compared to a compliant client, which typically connects to 50-100 peers. In § VI-C, we extend our free-riding study to further motivate our design. We experiment with free-riders in larger PlanetLab-residing torrents comprising of $\sim$400 leechers and under more realistic bandwidth distribution. We also investigate how the existence of seeders affects the effectiveness of the exploit.

Free-rider clients that employ the large view exploit are able to download faster than or almost as fast as its tit-for-tat compliant counterparts. In addition, as the number of free-riders increases, the swarm suffers performance degradation.

The exploit is more beneficial for free-riders when the swarm has many seeders, in which case some free-riders perform better than cooperative clients. When the swarm has no additional seeders other than the initial seeder, free-riders do not fare as well. However, they still attain good download rates compared to compliant clients, despite relying on downloading from the initial seeder and on the increased frequency with which they become optimistically unchoked. In all cases, the swarm suffers performance degradation as the number of free-riders increases.

Our results suggest that the large view exploit has the potential to be widely adopted because it is beneficial for free-riders. A dire prediction is that if more and more users that are reluctant to upload employ free-riding clients, BitTorrent communities will experience the "tragedy of the commons," until those users realize that they need to use cooperative clients in order to improve their download rates. Dandelion's non-manipulable incentives explicitly address this issue by preventing free-riders from obtaining any content without reciprocating or spending money.

The same weakness of BitTorrent's incentives is experimentally demonstrated in a recent work by Locher et al. [10]. In addition, Zghaibeh [24] reported that the portion of peers that free-ride and are able to attain good download rates (up to 300KB/sec) is already larger than expected (up to 10%).

III. Overview and System Model

We now provide an overview of Dandelion and describe the system model under which it is designed to operate. In addition, we introduce our setting and notation.

A. System Overview

When the content provider is overloaded, the Dandelion *server* redirects its *clients* to other clients that are able to serve their requests for content. The content provider splits content into verifiable *chunks*, and clients exchange carefully selected chunks. The content provider deploys in addition to the server, at least one client with the complete content (initial seeder).

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 trade-offs in selecting a chunk size in the case of static content distribution in § VI-B1.

Dandelion employs a hybrid incentive mechanism. In case a client has content that interests a peer, but that peer does not have content that interests the client, the system entices the selfish client to upload by rewarding the client with credit. The system also rewards a selfish client with credit when the peer is unable to reciprocate at the rate it downloads from the client. For example, selfish seeders would always be rewarded in credit. The server maintains the credit balance of each of its clients and converts credit to monetary rewards, such as discounts on paid content. To ensure that no user can be dishonest in the content-for-credit transactions, we employ a fair-exchange mechanism based on symmetric key cryptography. This mechanism requires the involvement of a trusted third party in each transaction. We refer to this mechanism for exchange of content uploads for credit as *credit-based exchange* and the chunks that are uploaded under this mechanism as *credit-traded*.

A Dandelion client employs a tit-for-tat mechanism when its peers can reciprocate with content of interest. That is, the client uploads content to a peer at the same rate that the peer uploads content to the client. However, a simple tit-for-tat scheme, such as BitTorrent's, is susceptible to the "large view" exploit. Free-riders that connect to many peers in their swarm can benefit

considerably by their peers' initial offers. To address this issue we employ an optimistic fair-exchange mechanism [14, 15] based on public key cryptography. Optimistic fair-exchange requires the involvement of a trusted third party only in case a peer misbehaves. We refer to the tit-for-tat mechanism as *TFT-based exchange* and the chunks that are uploaded under this mechanism as *TFT-traded*.

B. System Model

We assume two types of clients, which we define as follows:

- *Selfish* (rational) clients strategize based on a utility function that describes the cost they incur when they upload a chunk to their peers and when they pay credit to download a chunk. It also describes the benefit they gain when they are rewarded with credit for correct chunks they upload. A selfish client aims at maximizing its utility.

  A selfish client may consider manipulating the credit system in order to increase its utility by misbehaving as follows: a) upload no chunks to a peer, and yet claim credit for them; b) upload garbage either on purpose or due to communication failure to a peer, and yet claim credit or be reciprocated with valid content by the peer; c) download chunks from selfish clients, and yet attempt to avoid being charged or reciprocating with chunks; d) attempt to download chunks from selfish peers that are not interested in its content without having sufficient credit; and e) attempt to boost its credit by colluding with other clients or by opening multiple Dandelion accounts.
- *Malicious* clients may be faulty or strategize based on irregular utility functions, e.g. their utility increases by harming others, despite not obtaining credit or content. They misbehave as follows: a) attempt to make honest clients appear as malicious or dishonest, or attempt to cause them to be charged for chunks they did not obtain; b) 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) upload invalid chunks aiming at disrupting the distribution of content.

We assume that a selfish or malicious client cannot interfere with the IP 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.

C. Setting and Notation

Before we describe our design, we introduce the setting and notation.

We use $\langle X \rangle$ to denote the description of an entity or object, e.g. $\langle X \rangle$ denotes a client X's ID, while X denotes the client itself. H is a cryptographic hash function such as SHA-1, *MAC* is a Message Authentication Code such as HMAC [27], and i refers to a time period (epoch). By i_X we denote epoch i at client or server X. $MAC_K[X,Y]$ denotes the MAC of the concatenation of items X and Y, using the key K.

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 item it currently downloads or seeds, its credit balance, and the content it can access. The clients and the server maintain loosely synchronized clocks using standard techniques, such as the Network Time Protocol (NTP).

Every client A that wishes to join the network must establish a transport layer secure session with the server S, e.g., using TLS [28]. A client sends its ID and password over the secure channel. The server S generates a random secret key, denoted K_{SA}, which is shared with A. K_{SA} is also sent over the secure channel. In addition, every Dandelion client A obtains from the server a public/secret key pair (PK_A, SK_A) that is issued by the content provider. A's peers obtain the public key certificate signed from the server directly from A. $sign_A[X]$ denotes the public key signature of A on the item X, using A's secret key SK_A. $verify_A[sign_A[X], X]$ denotes the verification of A's public key signature on the item X, using A's public key PK_A. $verify[]$ returns a boolean value. K_{SA} and the public key pair are renewed upon epoch change.

The rest of the messages that are exchanged between the server and the clients are sent over an insecure communication channel (e.g. over 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 and the sequence number, 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 i changes.

$X \longrightarrow Y$:[name] Z, W denotes that the client or server X sends to Y a message of type *name*, which contains Z and W.

IV. Design

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

We introduce our credit-based exchange cryptographic protocol for the fair and non-repudiable exchange of content uploads for real monetary value. We also describe our hybrid incentive scheme, which combines the credit-based exchange with the TFT-based exchange. The TFT-based exchange allows peers to directly barter uplink bandwidth resources, and warrants the involvement of the trusted third party (TTP) only in case of exceptions. By combining the two cryptographic protocols, we reduce the load that the credit-based exchange protocol induced on the online TTP in the original Dandelion protocol [11].

Our design is based on the premise that although a low cost server may not have sufficient network I/O resources to

directly serve content to its clients under overload, [29, 30] it may have sufficient CPU, memory, and I/O resources to execute many symmetric cryptography operations, to maintain protocol state for many clients, to access its clients' protocol state, and to receive/send short messages. However, CPU, memory and I/O are still limited resources. Therefore we aim at making the design as efficient as possible. We also argue that content providers incur lower costs when they purchase the otherwise unused or altruistically offered uplink bandwidth of their clients than when they purchase bandwidth directly from access providers (§ IV-G).

A. Credit as Incentives

We aim at providing strong incentives for a selfish client to upload to a peer that does not possess content of interest or to a peer that is unable to upload as fast as the client uploads to it. To this end, we employ a cryptographic protocol to ensure the fair-exchange of content uploads for credit.

This 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, and as a credit bank maintaining records of the clients' credit balances. 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 a client B, B can determine its validity 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 server and the credit base are logical modules and can be distributed over a cluster (e.g. using consistent hashing based on client ID) to improve scalability and fault-tolerance.

Although, a single low cost server may scale to a few thousands of clients (§ VI-A), a well-provisioned content provider may purchase more bandwidth and employ server farms that consist of tens or hundreds of Dandelion servers. In this way, a well provisioned content provider may support hundreds of thousands of clients at a much lower cost than if the provider provided a significant portion of the required uplink capacity itself.

B. Credit Management

Dandelion's incentive mechanism creates a market, which enables a variety of application scenarios. Our protocol is intended for the case in which users maintain paid accounts with the content provider. The currency employed by Dandelion is directly mapped to real monetary value that customers introduce in the market by purchasing content. We employ real instead of virtual currency to eliminate depletion, inflation and starvation issues that plague typical virtual currency systems [31].

Selfish clients may sell upload service to peers that are unable to reciprocate with equally fast uploads. The content provider rewards uploaders with a credit value $\Delta_r > 0$ for the uploading of a chunk, which is fixed for every chunk and every client. Downloaders spend Δ_c credit units for each chunk they download. A client is awarded sufficient initial credit to download the complete paid content from its peers, without having to upload. In this way, slow uploaders do not face starvation and they are able to expend their credit at the rate needed to achieve their desired download rate.

The content provider redeems a client's accumulated credit for monetary rewards, such as discounts on content prices or service membership fees. We assume that the content provider prices chunk uploads appropriately to ensure that for the vast majority of clients, utility increases when they utilize their uplink in exchange for credit. We set $\Delta_r = \Delta_c$, so that two colluding clients cannot increase the sum of their credit by falsely claiming that they upload to each other. A client can acquire a chunk from a peer that is not interested in the client's content only if the client's credit is greater than Δ_c. We could alternatively set $\Delta_c > \Delta_r$, but this would penalize good uploaders who would not be able to recover the full monetary value of the amount of bandwidth they used by reciprocating with the equal amount of bandwidth.

A user cannot boost its credit by presenting multiple IDs (the Sybil attack [32]) 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.

C. Client Access Control

Before we present Dandelion's fair-exchange mechanisms, we describe how Dandelion enables the server and its clients to determine which clients are authorized participants. We also describe how clients obtain information about the content and the swarm. Figure 1 provides a high-level description of the client access control protocol, which is inspired by ticket-granting authentication schemes such as Kerberos [33], and we describe it in detail below.

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

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

Step 2: If B has access to F, the server chooses a short list of peers $\langle A \rangle_{\text{list}}$, among the ones that are currently in the swarm for F. The policy with which these peers are selected depends on the specifics of the content distribution system. Each list entry contains the ID of the peer and the peer's inbound Internet address. For every peer A in A_{list}, S sends a ticket $T_{SA} = MAC_{K_{SA}}[\langle A \rangle, \langle B \rangle, \langle F \rangle, t]$ to B, where t is the current timestamp. The ticket T_{SA} is only valid for a certain time length L_{peer} and allows 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, B requests a new T_{SA} from S. The ticket T_{SA} enables A to filter out service requests from misbehaving or unauthorized peers. 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 $\langle F \rangle$.

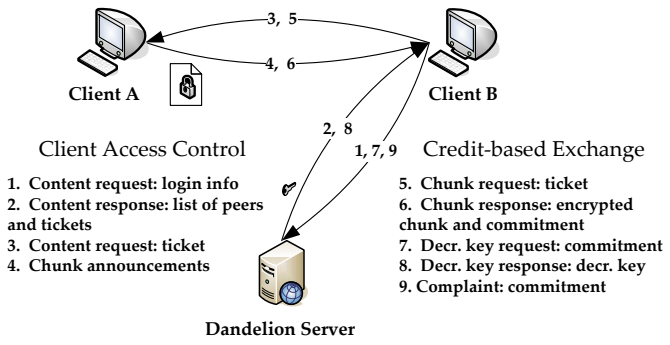

Figure 1: **The client access control and credit-based exchange protocols. Messages are listed in a "message type:contents" format and only the most important contents are included. The numbers on the arrows correspond to the listed protocol messages and the steps listed in Sections IV-C and IV-D.**

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

Step 3: Upon receiving the server's response, B connects to each client $A \in A_{\text{list}}$ to request the content $\langle F\rangle$. In the rest of this description, we list only the steps that involve B, and a specific client A.

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

Step 4: If *current-time* $\leq t + L_{peer}$ and T_{SA} is not in A's cache, A verifies whether $T_{SA} = MAC_{K_{SA}}[\langle A\rangle, \langle B\rangle, \langle F\rangle, t]$. If the verification fails, A drops this request. Also, if i_S is greater than A's current time period i_A, A learns that it should renew its key with S. Otherwise, A caches T_{SA} and periodically sends the chunk announcement message described below, for the period that 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 and how frequently depend on the type of content distribution.

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

D. Exchanging Content Uploads for Credit

We now describe in detail Dandelion's cryptographic credit-based exchange protocol (Figure 1).

Step 5: B and A determine which chunks to download from each other according to a chunk selection policy. For example, BitTorrent's locally-rarest-first [3] is suitable for static content distribution, while for streaming content or video on demand other policies are appropriate [13, 34]. 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] $T_{SA}, \langle F\rangle, \langle c\rangle, t, i_S$

Step 6: B's chunk requests are served by A as long as the timestamp t is fresh, and T_{SA} is cached or verifies. A encrypts c using a symmetric encryption algorithm Enc, as $C = Enc_{k_{\langle c\rangle}}(c)$. $k_{\langle c\rangle}$ is a key and encryption initialization vector pair generated as $(MAC_{K_{SA}}[\langle A\rangle, \langle B\rangle, \langle F\rangle, \langle c\rangle, t, 0]$, $MAC_{K_{SA}}[\langle A\rangle, \langle B\rangle, \langle F\rangle, \langle c\rangle, t, 1])$. Next, A hashes the ciphertext C as $H(C)$. Subsequently, it computes its commitment to the encrypted chunk as $T_{AS} = MAC_{K_{SA}}[\langle A\rangle, \langle B\rangle, \langle F\rangle, \langle c\rangle, H(C), t]$. The commitment T_{AS} is only valid for a certain time length L_{key}, which forces B to purchase the chunk at the server before T_{AS} expires. This fact allows A to promptly acquire credit for its service. Promptly acquiring credit may allow A to use the credit to download a file more than one time. It also allows B to provide chunks almost as soon as they are received, increasing global throughput at the early stages of the content's distribution. B has no real incentive to delay paying, since it has to do it eventually. He may attempt to refrain from uploading the content to speedup its download, but any gains would be cancelled out by the wait involved in uploading the content to earn the credit needed to obtain the decryption keys.

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

Step 7: Since B does not know the key K_{SA} that was used to generate $k_{\langle c\rangle}$ in step (6), it needs to request $k_{\langle c\rangle}$ from the server. As soon as B receives the encrypted chunk, B computes its own hash over the received ciphertext C' and sends a decryption key request message to S.

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

Step 8: If *current-time* $\leq t + L_{key}$, and the reported epoch of A is off by at most one, S checks if $T_{AS} = MAC_{K_{SA}}[\langle A\rangle, \langle B\rangle, \langle F\rangle, \langle c\rangle, H(C'), t]$. The commitment's 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 A or B and does not update their credit. S does not send the decryption key to B but it notifies B of the discrepancy. In case A repeatedly sends invalid chunk response messages, B is expected to disconnect from A and blacklist it. If B keeps sending invalid decryption key requests that involve A, S penalizes B. 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, S charges B and rewards A with Δ_c credit units. Subsequently, S computes $k'_{\langle c\rangle}$ as $(MAC_{K_{SA}}[\langle A\rangle, \langle B\rangle, \langle F\rangle, \langle c\rangle, t, 0]$, $MAC_{K_{SA}}[\langle A\rangle, \langle B\rangle, \langle F\rangle, \langle c\rangle, t, 1])$ 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), § IV-C), B complains to S by sending the following message.

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

S ignores this message if *current-time* $> t + L'_{key}$, where $L'_{key} > L_{key}$. $L'_{key} - L_{key}$ should be greater than the time needed for B to request and receive a decryption key response, decrypt the chunk and send a complaint to the server. With this condition, a misbehaving client A cannot avoid unfavorable complaint resolution by ensuring 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 L_{key}.

The client A cannot delay sending the complete chunk for time greater or slightly less than L'_{key}, because in such case B's decryption key request would not be considered valid by the server. Therefore, in such case B would never get to the complaint stage. T'-T should be such that even if the decryption key is received slightly before or after before T expires, B would still have time to decrypt the chunk and send a complaint to the server before T' expires. All that is required is that T'-T is greater than the time it takes to process the decryption key, decrypt the chunk and send a new complaint.

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* $> t + L'_{key}$.

If $T_{AS} \neq MAC_{K_{SA}}[\langle A\rangle, \langle B\rangle, \langle F\rangle, \langle c\rangle, H(C'), t]$, *S* punishes *B*. This is because *S* has already notified *B* that T_{AS} is invalid in step (8). If T_{AS} verifies, *S* caches this complaint, re-computes $k'_{\langle c\rangle}$ 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*, blacklists *A*, revokes the corresponding credit charge on *B* and notifies *B* that its complaint has been resolved. Similarly, *B* disconnects from *A* and locally blacklists it.

The server disconnects from a blacklisted client *A*, marks it as blacklisted in the credit file and denies access to *A* if it attempts to login. Future complaints that concern *A* and are non-duplicate, non-expired and with valid commitments, are ruled against *A* 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 local I/O error. Therefore, a client always verifies the read chunk against its hash before it encrypts the chunk and generates its commitment.

E. Rate-based Tit-for-Tat with Optimistic Fair Exchange

We now describe how Dandelion combines under one hybrid scheme the fair-exchange of content uploads for credit (credit-based exchange), with the tit-for-tat trading (TFT-based exchange) of content uploads.

The credit-based exchange addresses the fairness issue in tit-for-tat-based cooperative content distribution. Users that cannot upload at the rate their peers upload to them, pay the excess offered bandwidth in credit. However, when two clients are mutually interested in each others content and are able to upload to each other at the same rate, they are able to use tit-for-tat incentives, i.e. employ the TFT-based exchange. The TFT-based exchange aims at reducing the decryption key request load on the server. It enables a pair of clients to barter their uplink resources without requiring the server to mediate their transactions. At the same time, it prevents free-riding.

The *trade surplus* of a client with respect to a peer is the the difference between the number of TFT-traded chunks the client has uploaded to the peer and the number of TFT-traded chunks the client has downloaded from the peer. The trade surplus threshold ts is the maximum value that the trade surplus can take before the client with the positive trade surplus switches to credit-based exchange.

The trade surplus takes values greater than one when peers are not able to simultaneously exchange content and thus need more time to obtain chunks with which to reciprocate. In order to maintain the trade surplus in the presense of connection failures, the client keeps persistent session information for each of its peers, and identifies peers by their IP.

A strawman approach is to employ a tit-for-tat scheme under which clients exchange plaintext chunks, while ensuring that the trade surplus does not exceed a threshold [7, 35]. The problem with this scheme is that it cannot guarantee absolute fairness; the last peer that receives content may refrain from reciprocating. If in addition a free-rider employs the "large view exploit" or the trade surplus threshold is high, a free-rider can download a substantial amount of content without incurring any cost (see § VI-C). The free-rider problem is exacerbated under Dandelion, as the gains of free-riders translate to monetary loses for their peers. To address this problem, we employ an *optimistic fair-exchange* scheme that allows clients to barter their uplink in a fair manner. This scheme involves the server only in case of client misbehavior or communication failure. It is a purpose-built adaptation of classic optimistic fair exchange protocols [14, 15] and BAR Gossip [13].

If a client is a seeder (it does not expect its peers to obtain content that interests the client), it always uploads credit-traded chunks. Otherwise, if the client's trade surplus with a peer is less than or equal to the specified surplus threshold, each client responds to requests for chunks from its selected downloader peers with TFT-traded chunks. If the surplus exceeds *ts*, the client responds with credit-traded chunks.

When a client *A* uploads a TFT-traded chunk c_1 to a client *B*, c_1 is encrypted using symmetric key cryptography. To decrypt, *B* must reciprocate with at least one encrypted TFT-traded chunk c_2 to *A*. Upon reception of c', *A* sends the decryption key for *c* to B. In turn, *B* reciprocates with the decryption key for c_2.

If B sends an invalid chunk c_2 to A, A can detect it only after sending the decryption key for the valid chunk c1 to B. To address this issue, we include a non-repudiable complaint mechanism similar to the one used by the credit-based exchange. Unlike the credit-based exchange however, in the TFT-based exchange, senders commit to chunks and decryption keys they send using public key signatures. This is because the server is involved only in case of client misbehavior. Therefore, clients should be able to determine the validity of the commitments to the transmitted chunks without querying the server as in step (7) IV-D.

Note that public key operations are substantially more expensive that symmetric cryptography ones (§ VI-A2). Thus they are unsuitable for credit-based exchange, in which the server needs to validate the decryption key request for each uploaded chunk. In TFT-based exchange however, the server is involved only in case of complain

1) TFT-based Exchange

After two clients have authenticated each other using the client access control protocol, they may use TFT-based exchange. We now describe in detail Dandelions cryptographic TFT-based exchange protocol. For consistency with the previous description, we start the enumeration after step (4) of the client access control protocol.

Step 5: B and A determine which chunks to download from each other according to a chunk selection policy. B and A exchange requests for the missing chunks c_1 and c_2.

$B \longrightarrow A$:[chunk request] $T_{SA}, \langle F \rangle, \langle c_1 \rangle, t, i_S$

$A \longrightarrow B$:[chunk request] $T_{SA}, \langle F \rangle, \langle c_2 \rangle, t, i_S$

Step 6: A's and B's requests are served by each other as long as the timestamp t is fresh, and the tickets $T_{SA/SB}$ are cached or verify. A and B encrypt c_1 and c_2 using a symmetric encryption algorithm Enc, as $C_1 = Enc_{k_{\langle c \rangle}}(c_1), C_2 = Enc_{k_{\langle c \rangle}}(c_2)$. $k_{\langle c \rangle}$ is a key and encryption initialization vector pair generated as in step (6) of the credit-based exchange protocol. Next, A and B hash the ciphertexts C_1, C_2. Subsequently, they compute their commitment to the encrypted chunk using their public key. For example, A computes $TP_A = sign_A[H(\langle A \rangle, \langle B \rangle, \langle F \rangle, \langle c_1 \rangle, H(C_1))]$.

$A \longrightarrow B$:[chunk response] $TP_A, \langle F \rangle, \langle c_1 \rangle, C_1, i_A$
$B \longrightarrow A$:[chunk response] $TP_B, \langle F \rangle, \langle c_2 \rangle, C_2, i_B$

Step 7: Since A and B do not know the keys K_{SA} and K_{SB} used to generate the decryption keys $k_{\langle c_1 \rangle}, k_{\langle c_2 \rangle}$ in step (6), they need to obtain them from each other. As soon as A receives the chunk response from B,it computes its own hash over the received ciphertext C_2' and validates TP_B by computing $verify_B[TP_B, H(\langle B \rangle, \langle A \rangle, \langle F \rangle, \langle c_2 \rangle, H(C_2'), t)]$. If the signature TP_B is valid, A sends B the decryption key $k_{\langle c_1 \rangle}$. If the TP_B is invalid, A is expected to disconnect from B and blacklist it, in case B repeatedly sends invalid chunk responses. The same steps are taken by B when it receives the chunk response from A. When a client sends a decryption key for a TFT-traded chunk, it also includes a signature on the decryption key message. For example, client A computes $TPK_B = sign_A[H(B, \langle c_1 \rangle, k_{\langle c_1 \rangle})]$. This is to ensure that B can complain in case the client sends a valid TFT-traded chunk but does not send a valid decryption key, which would constitute a DoS attack.

$A \longrightarrow B$:[decryption key] $TPK_A, \langle F \rangle, \langle c_1 \rangle, k_{\langle c_1 \rangle}, i_A$
$B \longrightarrow A$:[decryption key] $TPK_B, \langle F \rangle, \langle c_2 \rangle, k_{\langle c_2 \rangle}, i_B$

We now describe the complaint mechanism for the TFT-based exchange.

Step 8: In case a client B receives an invalid TFT-traded-chunk and decryption key combination from A, it can use the server S to resolve complains. B sends to S A's signatures on the encrypted chunk (TP_A) and the decryption key (TPK_A) as well as the data that were signed (encrypted chunk hash, decryption key etc).

$B \longrightarrow S$:[complaint] $\langle A \rangle, \langle F \rangle, \langle c_1 \rangle, TP_A, H(C_1'), i_A$

S ignores the complaint message if a complaint for the same B and c is in a cache of recent complaints that S maintains for each client A or the reported epoch of A is off by more than one. Next, the server checks the validity of the signatures. If they do not verify, it blacklists B, because B could verify itself that they were invalid.

If they verify, the server caches the complaint and reproduces the correct decryption key $k_{\langle c_1 \rangle}$. It then determines whether $k_{\langle c_1 \rangle}$ matches the one signed by A. If it does not match, it blacklists B. If it matches, it retrieves the chunk c_1' from its local storage and encrypts it with $k_{\langle c_1 \rangle}$, $C_1'' = Enc_{k'_{\langle c_1 \rangle}}(c_1)$. If the hash of the ciphertext $H(C_1'')$ is equal to the value $H(C_1')$ that A sent to S, S decides that A has acted correctly and B's complaint is unjustified. Subsequently, S drops the complaint 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 and blacklists A.

The decryption keys for TFT-traded chunks cannot be retrieved from the server for a corresponding credit cost. In this way we discourage senders from not sending the decryption key for the last chunk in step (3) in an attempt to force their peers to pay them with credit instead.

2) Hybrid Incentives

ts: trade surplus threshold.
ts_p: trade surplus with peer p.
sec_p: number of sent encrypted chunks to peer p for which a decryption key has not been sent.
rec_p: number of received encrypted chunks from peer p for which a decryption key has not been received.
dks_p: decryption key surplus for p.
Input: The set of selected downloader peers P
Input: The set of the requested chunks from peer p that are available C

```
foreach peer p ∈ P do
    foreach chunk c ∈ C do
        if ts_p ≤ ts then
            SendTFTTradedChunk(c)
            sec_p = sec_p + 1
            ts_p = ts_p + 1
        else
            SendCreditTradedChunk(c)
        end
        if secp > 0 && dks_p < 1 && (rec_p > 0 ||
        dks_p = 1 ) then
            SendDecryptionKey(c)
            dks_p = dks_p + 1
            sec_p = sec_p − 1
        end
    end
end
```

Algorithm 1: The Hybrid incentive algorithm.

Algorithm 1 specifies how Dandelion combines the credit-based with the TFT-based exchange. Every client maintains the decryption key surplus dks_p for each of its peers p. dks_p is the difference between the number of decryption keys for TFT-traded chunks the client sent to p and the number of decryption keys the client received from p. A client sends to p a decryption key for a sent TFT-traded chunk if both of the following two conditions are satisfied: a) $dks_p < 1$; and b) it has received TFT-traded chunks from p for which it has not received decryption keys or $dks_p = 1$.

A Dandelion client must be able to switch between the credit- and the TFT-based exchange depending on content availability and peer upload and download rates. Therefore, we need an algorithm that aims at reducing the amount of creditexchanged chunks uploaded to each peer, while it ensures that the client uploads to its peers at the maximum rate its peers can download from it.

Rather than employing a complex per-peer resource allocation algorithm, we use the following simple scheme. At any moment a client selects a specified number n of peers to which to upload to using a downloader selection algorithm that is almost identical to BitTorrents. This algorithm aims at reducing the amount of content that is transmitted encrypted, and thus requires the involvement of the server. At the same time, it aims at increasing a clients uplink and downlink utilization. A Dandelion leecher ranks its peers based on the rate with which they upload TFT-exchanged chunks to the leecher. A Dandelion seeder ranks its peers based on the rate with which they download from the seeder, as is the case with BitTorrent seeders. Every time period T, the client selects as downloaders the *no* top ranked peers, and in addition it optimistically unchokes o additional peers for O seconds.

When a leecher selects the fastest TFT-exchanged chunk uploaders, it selects peers that are more likely to match its TFT-based upload rate and thus exchange chunks in a tit-for-tat fashion. This results in invoking the credit-based fair-exchange mechanism less often. A similar property has been shown to hold for BitTorrents chocking algorithm, which induces clustering between peers that have comparable upload rates [36, 37].

The trade surplus threshold values used by the peers can be changed locally and the server cannot enforce them. The server can only recommend values based on the swarms performance characteristics (see § VI-B1). Peers may choose to follow that recommendation or not. In general, if the recommendation for the TFT-exchange trade surplus is a low value, peers would have no problem complying. But if the recommendation is a high value peers may be reluctant to comply, because it would mean that they are more vulnerable to denial of service attacks, and to wasting bandwidth on encrypted chunk uploads that are not reciprocated. Still, clients are motivated to abide by the recommendation because compliant peers unchoke the fastest uploaders of TFT-traded chunks.

The decryption keys for TFT-traded chunks cannot be retrieved from the server for the corresponding credit cost. In this way we discourage senders from not sending the decryption key for the last chunk in step (3) in an attempt to force their peers to pay them with credit instead.

F. Design Properties

We now list the properties of our design.

Lemma 1 A selfish or a malicious client cannot assume another authorized client A's identity and issue messages under A. Thus, it cannot obtain service at the expense of A or cause A to be charged for service it did not obtain or cause 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 1 A misbehaving client can be successful in such attack only 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 2 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 2 B gets charged only if the commitment T_{AS} that S gets from B in step (7) is valid. This means that the values A sent to B in Step (3) are the ones A used to compute T_{AS} and that $H(C) = H(C')$. Since H is a second pre-image resistant cryptographic hash that B computes itself on C' received from A, $C = C'$. Thus, B is charged only if the encrypted chunk received by B is the encrypted chunk to which A has committed to. Since the same $k_{\langle c \rangle}$ is used by A to encrypt c into C' and by B to decrypt C into c', $C = C'$ implies that $c' = c$.

If A encrypts an invalid chunk c and sends it to B, B can issue a complain to S. For the complaint to be ruled against B, we should have $H(C') = H(C'')$, where C'' is computed by S in step (9). Since T_{AS} is accepted by S, all the values S used to compute T_{AS} are the ones that A sent to B, and the hash $H(C')$ is correctly computed over the ciphertext C that A sent to B. Consequently, S would generate the same $k_{\langle c \rangle}$ with the one A used. Therefore, S's encryption C'' would not be the same as the C that A sent to B. Consequently, $H(C')$ would not be equal to $H(C'')$ and S would reverse B's charge.

Lemma 3 If a selfish client A always encrypts chunk c anew when serving a request, as described in step (6) (§ IV-D), and if B gets a valid c from A, then A is awarded Δ_c credit units from S, and B is charged Δ_c credit units from S.

Proof 3 A generates $k_{\langle c \rangle}$ using a secure MAC function and a secret key K_{SA}, which is unknown to B because it was transmitted over the secure session between A and the server S. Therefore, the only way for B to retrieve $k_{\langle c \rangle}$ is to request it from S (step (7)), in which case S logs a charge against B.

Note that we are not concerned with another selfish client E being able to eavesdrop on the plaintext key $k_{\langle c \rangle}$ and the encryption of chunk c and thus retrieve c without being charged. In this case, the damage to the system is not significant because, A is still rewarded for its upload and second E is not consuming additional system resources.

The only way B could possibly avoid this charge is by sending a complaint to S, which includes T_{AS} and $H(C')$. For the complaint to be ruled in favor of B, it

should hold that $H(C') \neq H(C'')$, where C'' is computed by S in step (9). However, S will accept T_{AS} only if $T_{AS} = MAC_{K_{SA}}[\langle A\rangle, \langle B\rangle, \langle F\rangle, \langle c\rangle, H(C'), t]$. This means that all the values S used to compute T_{AS} are the ones that A sent to B, and that the hash $H(C')$ is correctly computed over the ciphertext C that A sent to B. Since this is the case, S would generate the same $k_{\langle c\rangle}$ that A used, hence S's encryption C'' would be the same as the C that A computed. Consequently, $H(C')$ would be equal to $H(C'')$ and B would be unable to reverse its charge.

Lemma 4 A malicious client cannot replay previously sent valid requests to the server or generate decryption key requests or complaints under another client A's ID. Thus, it cannot cause A to be charged for service it did not obtain or cause A to be blacklisted because of invalid or duplicate complaints.

Proof 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 i change. Thus, a malicious client cannot forge the source of the request, neither it can resent a request that has already been received.

Lemma 5 Under the TFT-based exchange scheme, a selfish client B cannot obtain a chunk c_1 from its peer A without expending bandwidth to reciprocate with a valid chunk c_2 itself. In addition, as stated in [13], a rational B prefers to send the short decryption key for an already sent valid chunk c_2, rather than repeatedly receive requests for the key from A.

Proof 5 Given that we use a secure MAC, the client cannot obtain the decryption key, unless it reciprocates with one valid TFT-encrypted chunk. A selfish client may attempt to transmit invalid content in order to obtain the key in case it does not have useful content. However a client that transmits invalid chunks is detected and penalized by the system through the complaint mechanism. Therefore a client is forced to upload a valid encrypted chunk. In such case, the selfish client has no incentive not to send the valid decryption key for the TFT-exchanged chunk, because the transmition of the key is a very cheap operation compared to the transmission of the chunk. In addition, a rational peer would prefer to send the key rather than receive repeated key requests, because the cost of sending the correct decryption key is less than repeatedly receive requests.

Observation 1 To maintain an efficient content distribution pipeline, a client needs to relay a 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 and verify its integrity prior to uploading the chunk to their peers. Therefore, homomorphic-encryption-based approaches or bundling many key requests together or re-encrypting the content and resending it before checking it are not appropriate.

Observation 2 If a client does not have sufficient credit, it cannot download chunks from a selfish peer that is not interested in the client's content. Our design choice to involve the server in each exchange of content uploads for credit instead of using the fair exchange technique proposed in [38], enables the server to check a client's credit balance, before the client retrieves the decryption key of a chunk.

In particular clients could abuse the Li et al. scheme [38] as follows. A user connects under A's account and downloads a certificate from the server, which indicates that it has enough credit. The certificate cannot indicate how much credit is left, because credit may increase. This user downloads the complete file and earns some credit in the process too. Then he gives A's account to his friend, which now has credit, thus it gets a certificate. The new user uses more than its available credit, issuing valid commitments (payment orders) signed with its new secret key, which the peers cannot verify whether they represent real value in the bank.

This problem can be solved if we disallow a specific user ID to download the same content more than once, no matter how much content they have uploaded. But this is not practical as the same user may choose to download the files it paid for multiple times from various machines, counting on some of his machines uploading a lot and gaining extra credit. We wish to avoid limiting how users can use their credit. If a user has paid for access to a file and he has sufficient credit he should be able to utilize the network's resources to download it again.

Observation 3 A malicious client B can abandon the credit-based exchange protocol after receiving the encrypted chunk without completing the transaction. In such case, A does not receive any credit, even though B has consumed A's resources. This is a denial of service (DoS) attack against A. Note that this attack would require client B to expend resources proportional to the resources of the victim A, therefore it is not particularly practical. Furthermore, we prevent clients that have been designated as misbehavers (blacklisted) in step (9) or clients that do not maintain paid accounts with the content provider from launching such attacks; the server S issues short-lived tickets T_{SA} (step (2), § IV-C) only to authorized and non-blacklisted clients. T_{SA} is checked for validity by A (steps (4) and (6) above).

Observation 4 A malicious client A may send a credit-traded chunk with an invalid MAC signature aiming at performing a DoS attack against B, without becoming blacklisted by the server. This attack would require client A to expend resources proportional to the resources of the victim B, therefore it is not particularly practical. In addition, a victim can be attacked by only one chunk before it locally blacklists the attacker. Furthermore as before, we prevent unauthorized clients or ones that have been blacklisted by the server from launching such attack.

Observation 5 A malicious client Ax may send a valid TFT-traded chunk to a client B but not release the decryption key, aiming at performing a DoS attack against B. Again, this attack would require client A to expend resources proportional to the resources of the victim B, therefore it is not practical.

If the attacker sends an incorrect decryption key, the server can arbitrate the complaint because the decryption key is signed. Subsequently, the server can blacklist the attacker and the attacker can no longer obtain tickets to contact more peers. In addition, the victims can be attacked by only one chunk before they locally blacklist the attacker. Furthermore as before, we prevent unauthorized clients or clients that have been blacklisted by the server from launching such attack.

Observation 6 A malicious client *A* may send a TFT-traded chunk with an invalid public key signature aiming at performing a DoS attack against *B*, without becoming blacklisted by the server. Again, this attack is not practical because it would require client *A* to expend resources proportional to the resources of the victim *B*. In addition, a victim can be attacked by only one chunk before it locally blacklists the attacker. Furthermore, we prevent unauthorized clients or clients that have been blacklisted by the server from launching such attack.

Observation 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 complaint resolution. That client must be a user registered with the system, otherwise it is not able to mint a complaint that merits resolution. Even if the client is a registered user, it becomes blacklisted by both the server and its peers the moment an invalid complaint is ruled against it. In addition, a malicious client cannot attack the server by sending valid signed messages with duplicate valid complaints. Our protocol detects duplicate complaints through the use of timestamps and caching of recent complaints.

Owing to Lemmas 1, 2, 3 and 5 as well as Observation 2, and given that the content provider appropriately valuates chunk uploads, Dandelion ensures that most selfish clients increase their utility when they upload correct chunks. At the same time, misbehaving clients cannot increase their utility. Consequently, Dandelion provides strong incentives for most selfish clients to upload to their peers.

G. Discussion

We now discuss our scheme's economic viability and potential for adoption. We argue that a content provider obtains more gains using our approach than by using a protocol such as BitTorrent that does not provide robust incentives for seeding. When seeding is not strongly incentivized, a content provider needs to purchase additional hardware and bandwidth to directly provide a large portion of the required upload capacity.

On the other hand, Dandelion enables the content provider to make a less expensive investment towards rewarding cooperative peers with real money. As a result, peers are strongly incentivized and the total upload capacity of the swarm increases.

Next, we support our insight that it is cheaper for content providers to purchase bandwidth from their users than purchase the infrastructure to directly serve content or purchase the service of third party CDNs. We make the conservative assumption that half of the price paid by broadband customers goes towards purchasing the uplink bandwith. Based on current DSL, Cable and FiOS offers in the US we extrapolate that user uplink bandwidth costs between $2 and $5 per Mbps per month [39, 40]. On the other hand, depending on location, it costs at least $40 to $80 per Mbps for a content provider to purchase T-3 to OC-12 bandwidth, with the cost of an OC-12 installation being on the order of $500000 [41]. We can also view uplink bandwidth costs on a per uploaded GB basis. Amazon's S3 Storage Service [42] valuates uplink bandwidth at approximately $0.18 per GB. On the other hand, a user that pays $5 per Mbps per month for his uplink may upload approximately 320GB in a month at a cost of approximately $0.015 per GB.

Therefore, although with Dandelion the content provider expends money to purchase its clients' bandwidth, he might incur lower cost compared to purchasing server bandwidth from Internet service providers. In addition since the content providers sets the prices for uploads he is in control of the peer bandwidth market, thereby he could reward users with less money than the actual cost of their uplink. Although a user's uplink bandwidth may cost more than the content provider is willing to pay, that bandwidth is typically unused or altruistically assigned to other P2P applications. Of course, the communication lines from ISPs are much more reliable, however the reliability issue is offset by the sheer number of client connections at the disposal of the content provider. Therefore, we hypothesize that our scheme can enable users to benefit from their spare bandwidth and content providers to tap into that relatively low cost resource. Validating this hypothesis requires a real market experiment, which is beyond the scope of this work.

We note that Dandelion's reward scheme is not the same as per-byte pricing per-byte or volume-based bandwidth pricing schemes. The arguments in favor of flat rate pricing are not directly applicable to our case because its characteristics are different to the Internet or cell phone pricing: in Dandelion users do not pay for every byte they receives/sends, instead they get a discount for every byte it serves. In our setting, users pay the flat rate for purchasing the content (which is exactly what many DVD rental or download services currently do), and our micropayments are used towards accumulating discounts. Unlike purchasing access to a communication service, users purchase the content (on a per-item or subscription basis) and accumulate credit to be used towards discounts. This business model works well with airline frequent flyer programs, with retail store reward cards etc. In constrast to the typical per-byte or volume-based pricing for communication services, our users know what is the maximum amount of money they will be charged and this is the rate for purchasing the access to the content. Thus, the user does not run the risk of being charged an unanticipated excessive amount of money or being denied further access to the service in case of overage.

Nevertheless, it remains an open question whether users will find Dandelion's pricing and discount model attractive, as there is limited prior experience with this model in the context of P2P content distribution.

As advocated in [31, 43], the proper market-based valuation of client bandwidth resources according to demand and

content rarity is also an important issue, which we are not addressing in this work. Nevertheless, our non-manipulable hybrid incentive design can be integrated with schemes that allow variable pricing of resources to address market-based incentive issues in a distributed or centralized way [43, 44]. Such integration would require changes to the chunk and peer selection algorithms, as well as the initial assignment of credit to new clients. Such algorithms should enable peers to choose among the resource offerings based on individual chunk prices and the offered transfer rates. In this work, we instead focus on the problem of ensuring the fair-exchange of payment for content. Practice has shown that this problem is a major stumbling block to the commercial adoption of micropayment-based incentive schemes [12].

Our solution breaks the barrier to entry for small content providers (e.g indie movie studios) and leads to a more open and competitive market for Internet resources. A content provider no longer has the limited choice between heavily investing in infrastructure or buying third party services (Akamai, BitTorrent inc) to bootstrap. Instead of paying a substantial initial amount to over-provision hardware and bandwidth, the content distributor can pay solely for the bandwidth its clients have actually contributed. Moreover, the content provider itself is in the advantageous position of determining the price with which it purchases its client's bandwidth. Even well-provisioned content distributors may use Dandelion to further save on bandwidth costs. This is illustrated by the relative success of Roo's Peer Delivery Network [45] based on Peer Impact [46]. We improve upon Roo's PDN by offering non-manipulable currency. Roo has not publicized their micropayment scheme but it is very likely that it is not cryptographic.

Our solution can also be deployed by commercial Content Distribution Networks to promote cooperation in peer-assisted content distribution. CDN providers are considering the use of peer-assisted content distribution, in order to offer new lower tier CDN services at reduced costs. This is illustrated by the recent purchase of "Red Swoosh" by Akamai [47].

H. Discouraging Unauthorized Content distribution

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 incentive to upload to a 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 employ the fair-exchange protocol, thus the server is able to not reward a client that serves unauthorized peers.

Clients are able to verify the legitimacy of requests for service (steps (1) and (5)), hence they can avoid wasting bandwidth to serve unauthorized clients. Furthermore, precisely because of this ability, clients can be held legally 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 even more appealing to distributors of copyright-protected digital goods.

For example, selfish seeders have no incentive to facilitate unauthorized content distribution. Our scheme motivates seeders to behave selfishly and discourages them from behaving altruistically. That is, seeders are more reluctant to waste bandwidth to upload to unauthorized users when they can use their bandwidth to upload to authorized users and earn monetary rewards instead. This phenomenon has been empirically observed in various social settings by Frey et al [48] and has been termed the crowding-out effect: the presence of extrinsic motivations (such as financial rewards) results in decreased intrinsic motivation (such as ideological altruism).

In some BitTorrent deployments, content access policies are enforced by requiring 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 seeders, which are by definition altruistic and have no real motivation to deny service to unauthorized peers. Consequently, a single authorized, rational but misbehaving peer can facilitate illegal content replication at a large scale.

V. Implementation

This section describes a prototype C implementation of the Dandelion system, which is suitable for static content distribution. It uses the *OpenSSL* toolkit for cryptographic operations.

Our initial approach was to build Dandelion on existing BitTorrent codebase. However, we quickly realized that it is preferably to build our protocol from scratch, as it requires extensive modification of all primary BitTorrent functions. First, the Dandelion server, in addition to swarm view tracking, performs processing for the fair-exchange protocol. Second, unlike BitTorrent trackers, a Dandelion server may be CPU or disk I/O bound instead of network I/O bound, depending on the capacity of the access link, therefore different performance optimization strategies are warranted. Third, the Dandelion client performs additional processing for the fair-exchange protocol. Fourth, the Dandelion client does not employ rate-based tit-for-tat, and thereby its downloader selection (unchoking) mechanism is substantially different.

A. Server Implementation

For simplicity, our current implementation combines the content provider and the credit management system at a single server. It is our future work to scale the Dandelion server by balancing its load over multiple machines. For example, by having dispatchers with high downlink bandwidth redirecting requests to the Dandelion servers hosts that are responsible for the client IDs involved. The assignment of client IDs can be done using consistent hashing.

Our current server implementation is single-threaded and event-driven. The network I/O 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 dispatching mechanism. In our implementation `epoll` is used as level-triggered, while its use as edge-triggered could further improve performance The server stores

in heap memory information for each of the clients with which it has an active Dandelion session.

The server uses standard file I/O system calls to efficiently manage persistent client information, which is stored in a simple file called the credit file. Each client is assigned an entry in the credit file, which keeps the client's credit, its authentication information and its 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 credit file is sufficiently large to accomodate as many client entries as may be needed.

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 asynchronously 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, and 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 (operation 14 in Table I, § VI-A2).

Nevertheless, in the face of frequent system failures, we can avoid the performance penalty of per-transaction commitments to persistent storage and still provide satisfactory data safety guarantees. This can be done by assigning to a helper process the task of periodically synchronizing the credit file with persistent storage (§ VI-A2).

B. 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 a specified number of peers (typically 50-100). With uploader discovery, a client strives 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 function of announcing recently received chunks, requesting missing chunks, requesting decryption keys for received encrypted chunks, and replying to chunk requests. Our implementation employs rarest-random-first [49] scheduling in requesting missing chunks from clients. To efficiently utilize their downlink bandwidth, clients dynamically adjust the number of outstanding chunk requests r that have been sent to a peer and have not been responded to [3, 49]. r depends on the observed download rate from the peer and the time between a request for a chunk is sent and the complete chunk is received according to the equation $r = turnaroundTime \times peerDownloadRate/chunkSize$. We described the downloader selection algorithm in § IV-E.

VI. Evaluation

The goals of this experimental evaluation are: a) to identify the scalability limits of Dandelion's centralized non-manipulable virtual-currency; b) to examine the trade-off between performance and scalability in selecting the chunk size and trade surplus threshold; c) to motivate our design by demonstrating the importance of incentives for seeding and the impact of free-riding in BitTorrent-like swarms; and d) compare the performance of our Dandelion-based static content distribution system to BitTorrent. [1]

A. Server Performance

In this section, we evaluate and profile the server in terms of decryption key and complaint request throughput.

1) Server Throughput

A Dandelion server mediates the chunk exchanges between its clients. The client plaintext download throughput and the scalability of our system is bound by how fast a server can process their decryption key requests (step (8), § IV-D). Both the server's computational resources and bandwidth may become the performance bottleneck. We deploy a Dandelion server on a dual Pentium D 2.8GHZ/1MB CPU with 1GB RAM and 250GB/7200RPM HDD running Linux 2.6.5-1.358smp, which shares a 100Mbps Ethernet II link. To mitigate bandwidth variability in the shared link and to emulate a low cost server with uplinks and downlinks that range from 1Mbps to 5Mbps, we rate-limit our Dandelion server at the application layer. We deploy ∼1000 clients that run on ∼100 distinct PlanetLab hosts.

The clients send requests for decryption keys to the server and we measure the aggregate rate with which all clients receive decryption key responses. The server always queries and updates the credit record from and to the credit file without forcing commitment to disk. We run each experiment for a specified per-client request rate, which varies from 1 to 6 req/sec. For each request rate, the experiment duration is 10 minutes and the results are averaged over 10 runs.

Figure 2 depicts the server's decryption key response throughput for varying server bandwidth. As the bandwidth increases from 1Mbps to 4Mbps, the server's throughput, indicating that for up to 4Mbps access link, the bottleneck is the bandwidth. For 5Mbps and 4Mbps the throughput is almost equal, indicating that for 5Mbps the bottleneck is the CPU. The results show that a server running on our commodity PC with 4Mbps or 5Mbps access link can process up to ∼3105 decryption key requests per second. This result suggests that with a 256KB chunk size, this server may simultaneously support almost 3100 clients that download only credit-traded chunks at 256KB/s. With a larger chunk size and TFT-based

[1] Dandelion's source code for Linux and scripts to run our experiments can be downloaded at http://www.ics.uci.edu/~msirivia/dandelion.

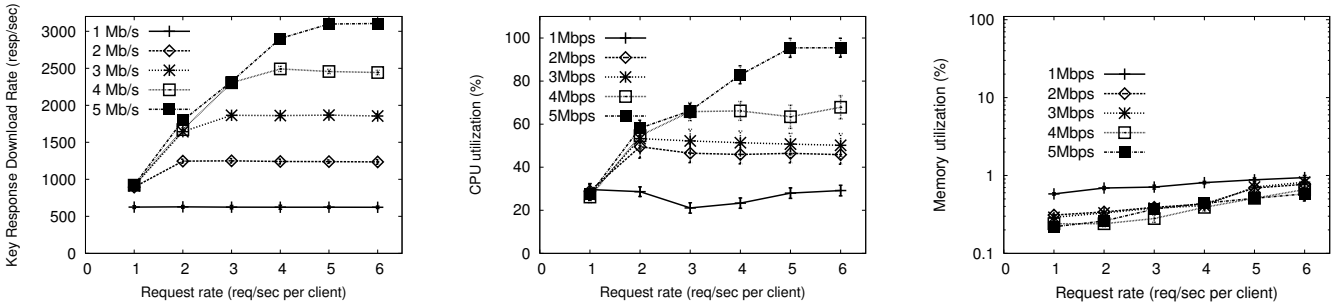

Figure 2: **Server decryption key response throughput evaluation. as a function of specified per-client key request rate, for varying server rate-limits. ∼1000 clients send decryption key requests: (a) aggregate decryption key response throughput of the Dandelion server; (b) server's CPU utilization(%); (c) server's memory utilization (% in logarithmic scale).**

exchange, each such client receives credit-traded chunks at a lower rate. Thus, the number of supported clients increases.

We also run experiments with ~ 500 clients each set to send 10 req/sec (not depicted in Figure 2). By comparing the throughput of the 5Mbps server's throughput in this case (3114 req/sec) with its throughput when ~ 1000 clients send 5 req/sec each (3105 req/sec), we infer that the server's throughput is independent of the number of clients since the epoll performance in the examined range of number of connections is independent of the number of connected clients.

From Figures 2(b),(c) we make the following observations with respect to CPU and memory utilization at the server when ∼1000 clients send 6 req/sec. For 5Mbps, the server's CPU utilization reaches ∼100%, indicating the CPU as the bottleneck. We also observe that the server consumes less than 1% of the available memory, even under overload.

A Dandelion server is also responsible for resolving complaints (step (9), § IV-D). A complaint resolution involves the expensive disk I/O operation for reading a chunk. Therefore it represents a performance bottleneck in case the system receives too many complaints. We performed an experiment with ∼1000 clients sending 5 decryption key req/sec, and 15 clients sending 1 complaint resolution request per sec (involving a randomly selected 256KB chunk of a 1GB file). The 5Mbps server was able to deliver roughly 1490 decryption key responses per second along with 14 complaint resolution responses per second. In another experiment with ∼60 complaint sending clients, we measured the maximum complaint resolution throughput of a 5Mbps Dandelion server to be 52 complaint responses per second.

Note that the server does not need to deliver high complaint resolution throughput for the reasons listed in Observation 6, § IV-F. First, complaints can involve only clients that are registered with the system. Second, once a complaint has been resolved against a client, that client becomes blacklisted and all future complaints concerning the misbehaver are automatically resolved against him without performing the expensive cryptographic and disk I/O operations. In addition, to improve throughput, the expensive disk I/O operation can be performed in parallel with the decryption key request processing using asynchronous I/O.

2) Server Profiling

We profile the cost of operations at the server aiming at identifying the performance bottlenecks of our design. We use the same machine as the one used in the previous section.

Table I lists the cost of Dandelion operations. Timings for operations 1-4 and 6-8 are obtained using `getrusage()` over 10000 executions. Timings for operations 5, 14, 15 and 16 are approximated using `gettimeofday()` over 10000 executions. Operation 5 reads from the disk a new randomly selected 256KB chunk of a 1GB file in each execution. Operations 14-16 are performed on a credit file with 10000 44-byte entries. Timings for operations 11-13 are approximated according to our application layer rate-limiting for 5Mbps uplink and downlink. They are provided as reference for comparison with CPU-centric and credit management operations. Operation 6 uses 8-byte-block Blowfish-CBC with 128-bit key and 128-bit initialization vector, with at most 8 byte padding. Operations 1-4 use HMAC-SHA1 with 128-bit key. Operation 7 uses SHA-1. Operation 8 uses 1024-bit RSA signatures. For operation 14, we use `fsync()` and we disable HDD caching.

The main tasks of a Dandelion server are to: a) receive the decryption key request (operation 11); b) authenticate the decryption key request (operation 1); c) verify the commitment (operation 2); d) compute the decryption key (operation 3); e) query and update the credit of the two clients involved (operations 14 and 15); f) sign the decryption key response (operation 4); and g) send the decryption key response (operation 12).

The signed decryption key request and decryption key responses are sent over an insecure TCP connection. A client establishes and uses the secure TLS channel with the server (operation 10) only to send authentication information (once per Dandelion session), the shared key and the public key pair (the same keys are used for a relatively long period).

As can be seen in Table I, the per-decryption-key-request cryptographic operations of the server (operations 1-4) are highly efficient (total 12 μsec), as only symmetric cryptography is employed. The credit management operations (14 and 15) are also efficient (total 24 μsec). The communication costs of receiving and sending decryption key responses (operations 11-12) are clearly higher than the cryptographic computation costs. In addition, operations 11-12 can take place concurrently with each other and the computational operations.

	Dandelion operation	Size	Time (ms)
	CPU-centric Operation		
1	Authenticate decryption key request	58 bytes	.003
2	Generate symmetric cryptography commitment for decryption key request or complaint verification	38 bytes	.003
3	Compute decryption key	19 bytes	.003
4	Sign decryption key response	46 bytes	.003
5	Read chunk	256 KB	31
6	Encrypt chunk	256 KB	2.876
7	Hash encrypted chunk	256 KB	1.017
8	Verify public-key commitment for complaint verification	128 bytes	.2
9	Event dispatching `epoll_wait()`) on 1000 socket descriptors	N/A	0.002
10	Establish SSL session (`SSL_accept()`)	N/A	19
	Communication Operation		
11	Receive decryption key request	96 bytes	∼.26
12	Transmit decryption key response	84 bytes	∼.24
13	Receive TFT-based exchange complaint	204 bytes	∼.55
	Credit Management Operation		
14	Query credit file	N/A	∼0.004
15	Update credit file without commit to disk (rely on OS)	N/A	∼0.02
16	Update credit file and force commit to disk	N/A	∼9.25

Table I: **Timings of Dandelion operations.**

Committing the credit file per transaction (operation 16), could yield 10-20 times lower decryption key response throughput than relying on the OS to commit credit file updates (operation 15).

The cost of a complaint is substantially higher because in addition to receiving the message (operations 11 or 13), authenticating it and verifying a commitment (operations 2 or 8), it involves reading a chunk (operation 5), encrypting it with the sender's key (operation 6), and hashing the encrypted chunk (operation 7).

The cost of event dispatching (operation 9) is not significant if we use the highly efficient and scalable `epoll` instead of the `select` API. Indicatively, the less scalable `select()` costs 0.009 ms when used to dispatch 500 sockets and 0.017 ms when used to dispatch 1000 sockets. On the other hand, `epoll_wait()` costs approximately only 0.002 ms for both 1000 and 100 sockets. This reveals that the choice of event dispatching mechanism is critical for the performance of the system. Under the same experimental configuration with the one used in § VI-A1, a 5Mbps server that used `select()` instead of `epoll_wait()` was able to process only 799 decryption key requests per second.

B. System Performance

In this section, we experimentally evaluate the behavior of the entire Dandelion system on PlanetLab. We examine the impact of chunk size and the trade surplus threshold on the performance of the system. In addition, we demonstrate the performance gains of providing incentives for seeding. Last, we compare our system's performance to BitTorrent's. In all experiments we run a Dandelion server on the same machine as the one used in the previous sections, and the server is rate-limited at 5Mbps.

Leechers are given sufficient initial credit to completely download a file, according to the credit management policy discussed in § IV-B. Clients always respond to chunk requests from their selected downloaders. We also set the TCP sender and receiver buffer size equal to 120KB in order to cover the bandwidth-delay product. Both BitTorrent and Dandelion clients determine the number of outstanding chunk requests as described in § V-B.

In addition, we have adjusted system parameters such as number of unchoked peers and timeouts to achieve good performance under our bandwidth distribution in the PlanetLab environment. In particular, we set the parameters described in § IV-E as follows: T=10, n=10, o=2 and O=30.

We aim at making our evaluation representative of real Internet peer-to-peer content distribution swarms, while including as many PlanetLab nodes as possible. To this end, we partially emulate a typical client uplink bandwidth distribution [50] (Table II) by applying per-client application layer rate-limiting. To deal with PlanetLab hosts that are are unable to achieve the bandwidth values specified by that distribution, we impose a capped bandwidth distribution that does not faithfully reflect the one reported in [50]; it excludes nodes with very high upload capacity (greater than 350KB/sec).

Since Dandelion aims at enabling content providers to purchase end-user bandwidth, we expect Dandelion users to connect to the Internet via privately owned residential broadband links. It is reported that these links currently offer at most ∼3Mbps uplink capacity [51]. Nevertheless, we argue that the capped bandwidth distribution enables us to derive conclusions on the behavior of real P2P swarms that do not include very high capacity peers. In addition, as network administrators of large organizations, such as academic institutions, become increasingly concerned with resource consumption and copyright issues, the number of very high capacity academic (or other) nodes that participate in P2P content distribution may decrease substantially. In addition, we impose a download rate distribution to approximate the effect of asymmetric broadband links. Clients with less than or equal to 70KB/sec upload rate, are assigned a maximum download rate that is 5 times higher than their maximum upload rate. The rest of the nodes are assigned a download rate equal to 350KB/sec.

In particular, we observe that the maximum upload rate of 80% of hosts in the distribution reported in [50] is less than or equal to 350KB/sec. The remaining 20% of hosts require upload rates between 350KB/sec and 10000KB/sec. We periodically use Dandelion to distribute a 100MB file to ∼500 non-rate-limited hosts and we identify ∼400 nodes that are able to attain upload rates equal to or higher than 350KB/sec. For the 20% of hosts, that require bandwidth between 350KB/sec and 10000KB/sec, we were unable to identify a sufficient number of hosts able to consistently achieve such high rates. Thus, we assigned 350KB/sec upload rates to those nodes. The initial seeders in each experiment are rate-limited at 250KB/sec.

For each configuration we repeat the experiment 10 times and we extract mean values and 95% confidence intervals over the swarm-wide *download completion times*.

We note that we evaluate a particular implementation of

Portion of nodes	0.05	0.05	0.1	0.1	0.1	0.1
Bandwidth (KB/sec)	40	50	55	60	65	70
Portion of nodes	0.05	0.1	0.05	0.05	0.05	0.2
Bandwidth (KB/sec)	75	100	150	200	250	350

Table II: **Distribution of upload bandwidth of Dandelion and BitTorrent peers as used in our PlanetLab experiments. This distribution draws from the one reported in [50], but due to PlanetLab bandwidth constraints we omit hosts with upload capacity higher than 350KB/sec.**

Dandelion that is suitable for static content distribution. Although our results would vary for other P2P content distribution applications that use different chunk scheduling and peer selection policies, we expect our results to be qualitatively similar, allowing us to extract generic trends in the behavior of such systems. In particular we expect seeding to be beneficial and the chunk size and trade surplus threshold to affect performance, regardless of the specifics of the content distribution system.

1) Selecting Chunk Size and Trade Surplus Threshold

With this series of experiments we examine the trade-offs involved in selecting the size of the chunk and the trade surplus threshold of the TFT-based exchange. In addition, we motivate our hybrid incentive mechanism by quantifying its improvement in scalability over the credit-based-exchangeonly scheme proposed in [11].

Intuitively, since clients are able to serve a chunk only as soon as they obtain it, 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; 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. Also, due to TCP's slow start, a small chunk size cannot ensure high bandwidth utilization during the TCP transfer of any chunk. Last, small chunks yield increased control overhead.

In addition, under our optimistic fair exchange scheme, a receiver is able to acquire a TFT-traded chunk only after it reciprocates with a chunk of equal size and retrieves the decryption key. The larger a received TFT-traded chunk is, the longer the receiver may have to wait until it is able to respond with an equally large chunk. Only after decrypting the chunk the receiver is able to relay it to its peers, therefore a large chunk decreases the efficiency of the distribution pipeline.

As the trade surplus threshold ts and the chunk size increases, trading flexibility also increases. This enables a client to upload TFT-traded chunks in case its peers cannot temporarily match the client's upload rate. This results in reduction of the rate with which decryption key requests are sent to the server. However, a large threshold and chunk size may result in clients wasting bandwidth to transmit encrypted chunks that are never reciprocated and decrypted, causing performance degradation.

We use as performance metrics the mean download completion time of the clients and the decryption key request load on the server. In each configuration, we deploy approximately $\sim$400 Dandelion leechers and one initial seeder. Leechers start downloading the file almost simultaneously emulating a flashcrowd. The duration of each experiment is 2200 sec.

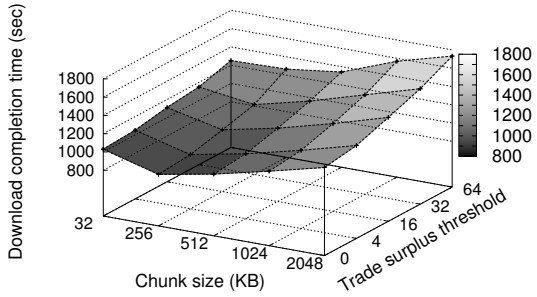

Figure 3: **Swarm-wide mean download completion times of $\sim$400 leechers as a function of chunk size and trade surplus threshold ts for a 100MB file. Both the z axis and the gray map depict the download completion time. The 95% confidence intervals (not depicted) take values between 50 and 100 sec.**

Figure 3 shows the leecher mean download completion time as a function of the chunk size and the trade surplus threshold. We observe that for larger than 256KB chunks, the system's performance degrades as the chunk size and the trade surplus threshold increases. For example, for $s = 0$, a 256KB chunk size yields better performance (864sec) than a 2048KB chunk size (1263 sec). 256KB chunks guarantee that there are sufficiently many distinct chunks for peers to exchange. The beneficial impact of smaller than 256KB chunks in terms of chunk scheduling flexibility is negated by the performance-degrading TCP effects and the increased control overhead. For example, for $ts = 0$, a 256KB chunk size yields notably better performance (864 sec) than a 32KB chunk size (1031 sec). In addition, we observe that the mean download completion times consistently increases with ts and that the degrading impact of ts on performance increases with the chunk size.

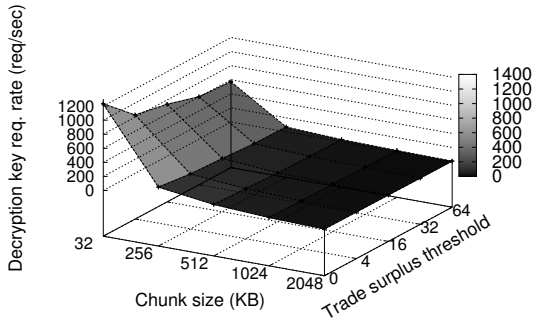

Figure 4: **The mean decryption key request rate at the server as a function of the chunk size and the trade surplus threshold ts for a 100MB file. Request rates are extracted in 10 sec intervals over the duration of the experiment. The 95% confidence intervals (not depicted) take values between 3 and 30 req/sec.**

In Figure 4, we observe that the load on the server decreases as ts increases. In particular, under our network configuration the decryption key request load decreases by approximately 40% when the system uses $ts = 16$ instead of $ts = 0$. At the

same time, the swarm-wide performance degrades only by 9 to 13%, depending on chunk size. Setting $ts = 0$ corresponds to using only credit-based exchange as was originally proposed in [11], while $ts > 0$ allows clients that are mutually interested in each others content to exchange chunks without involving the server. This result demonstrates the effectiveness of our hybrid incentive scheme in improving scalability by reducing the servers decryption key request load. As expected, the server load also decreases as the chunk size increases. The decryption key request load for 32KB chunks varies in ∼600 to ∼1200 req/sec depending on ts. For 256KB chunks it varies in only ∼70 to ∼190 req/sec. The evaluation for 32KB chunks enables us to roughly predict the load on the server in a swarm that consists of 8 times more clients but uses 256KB chunks

In Figure 4, we observe that the load on the server decreases as *ts* increases. In particular, under our network configuration the decryption key request load decreases by approximately 40% when the system uses $ts = 16$ instead of $ts = 0$. At the same time, the swarm-wide performance degrades only by 9 to 13%, depending on chunk size. Setting $ts = 0$ corresponds to using only credit-based exchange as was originally proposed in [11], while $ts > 0$ allows clients that are mutually interested in each others content to exchange chunks without involving the server. This result demonstrates the effectiveness of our hybrid incentive scheme in improving scalability by reducing the servers decryption key request load.

As expected, the server load also decreases as the chunk size increases. The decryption key request load for 32KB chunks varies in ∼600 to ∼1200 req/sec depending on *ts*. For 256KB chunks it varies in only ∼70 to ∼190 req/sec. The evaluation for 32KB chunks enables us to roughly predict the load on the server in a swarm that consists of ∼8 times more clients but uses 256KB chunks

For this particular swarm configuration, the content provider may determine that a 256KB chunk and *ts*=4 is a good configuration. It yields a low download completion time (893sec) and a relatively low server load (132req/sec). Unless mentioned otherwise, in the rest of this evaluation we use these values.

2) Impact of Seeders

Dandelions credit-based exchange mechanism strongly incentivizes clients to remain online after download completion, increasing the number of seeders in a swarm. With this series of experiments, we motivate our credit-based exchange mechanism by demonstrating the performance gains by the existence of additional seeders.

Intuitively, since typical P2P clients reside behind asymmetric links, content distribution swarms are expected to benefit by the existence of additional seeders. Seeders complement the swarm's uplink bandwidth without expending its downlink bandwidth. We demonstrate the impact of seeders in BitTorrent-like swarms by varying the probability that a leecher remains online to seed a file after it completes its download. Upon completion of its download, each leecher stays in the swarm and seeds with probability *a*, which varies in 0% to 100%. Leechers start downloading the file immediately upon arriving in the swarm. The duration of each experiment is 2200 sec.

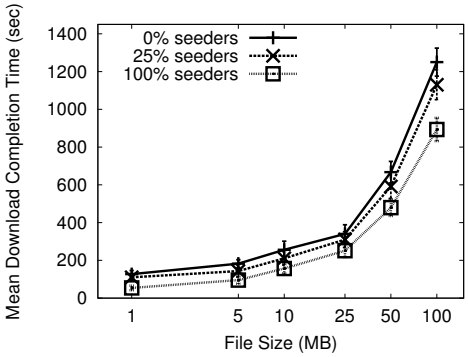

Figure 5: **Swarm-wide mean download completion times of ∼400 leechers as a function of file size for varying portion of leechers that become seeders. Clients arrive almost simultaneously.**

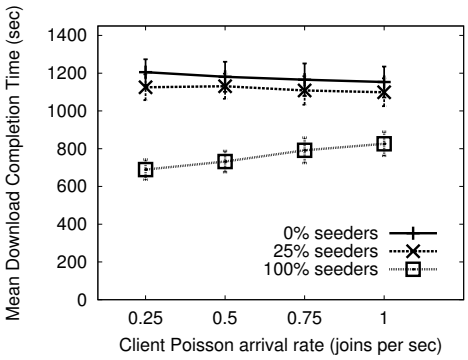

Figure 6: **Swarm-wide mean download completion times of ∼400 leechers as a function of the client Poisson arrival rate λ for varying portion of leechers that become seeders. Clients download a 100MB file.**

Figure 5 depicts the mean download completion time over ∼400 leechers as a function of the file size, for varying *a*. All clients join the swarm almost simultaneously. We vary the file size to demonstrate that the impact of seeding depends on the duration of the download, and to demonstrate the behavior of the system under different workloads. Our results show the beneficial impact of seeders. For example, for a 100MB file, we observe a swarm-wide mean download completion time of 893 sec and 1250 sec when $a = 100\%$ and $a = 0\%$, 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% or *a*=0%, we observe that the impact is reduced as the file size increases. We observe that as the file size decreases, the decrease of *a* causes a more dramatic increase of download completion times. The larger the file is, the longer leechers remain online to download it, thus they upload to their peers for longer periods. For smaller files however, peers have to rely heavily on leechers that become seeders.

We also evaluate the system under varying peer arrival patterns. We vary the Poisson parameter λ under which new clients join the swarm. Depending on the arrival pattern, seeders may play a more or a less beneficial role. For example, during a flash crowd (high λ) many peers finish their download at approximately the same time and therefore do not benefit each other when they remain online as seeders. As λ decreases,

new peers can benefit by more older peers that finish their download and remained seeding. Figure 6 depicts the mean download completion time over all ∼400 leechers as a function of the client Poisson arrival rate λ, for varying *a* and a 100MB file. The results show that seeders substantially benefit swarms with low arrival rates, as new peers take advantage of the additional uplink capacity of peers that arrived earlier and became seeders. For example, for $\lambda = 0.25$, we observe a swarm-wide mean download completion time of 689 sec and 1125 sec when $a = 100\%$ and $a = 0\%$, respectively.

3) Comparison with BitTorrent

Unlike BitTorrent, Dandelions incentive mechanism requires the involvement of a centralized component, uses optimistic fair-exchange of content uploads, employs a modified downloader selection algorithm and does not employ subpiecing [3]. In this section, we show that these differences do not have a negative impact on download completion time.

To this end, we compare the performance of a swarm of Dandelion clients with a swarm of BitTorrent (CTorrent DNH-3.2) clients. In both swarms, there are ∼400 leechers and one initial seeder, and leechers stay online to seed after download completion. Dandelion clients employ both the credit-based and TFT-based exchange protocols.

Figure 7 presents the CDF of the download completion times for both BitTorrent and Dandelion clients for a 100MB file. This illustration shows that a Dandelion swarm can attain performance comparable to a BitTorrent one, when both swarms have the same number of seeders. Although our Dandelion implementation appears to outperform BitTorrent, we do not claim that a Dandelion-based static content distribution system is better-performing. The performance of both protocols is highly dependent on numerous parameters, which we have not exhaustively analyzed. To name a few, such are the chunk size, the number of peers, the number of unchoked peers, the interval between unchoked peer set updates, number of pending chunk requests and the TCP sender and receiver buffer size. These parameters need to be fine-tuned according to factors such as swarm size, client bandwidth or expected RTT. For Dandelion, we have empirically fine-tuned these parameters, however for CTorrent, barring TCP sender and receiver buffer size, we have not tuned any other parameter.

C. Free-riding in BitTorrent-like Swarms

In this section, we provide additional motivation for the use of non-manipulable cryptographic fair-exchange incentives. We demonstrate that under BitTorrent-like incentives, free-riding is beneficial for free-riders and harmful for cooperative clients.

For all experiments we use a Dandelion implementation in which we disable the cryptographic fair-exchange protocols. With disabled fair-exchange protocols, Dandelion's implementation is almost identical to BitTorrent's. We use this implementation because it includes a trade-surplus mechanism and we have also validated it against BitTorrent (§ VI-B3). That is, it employs rate-based tit-for-tat and random-rarest-first chunk scheduling. We deploy ∼400 leechers and one initial seeder. All clients join the swarm simultaneously to download

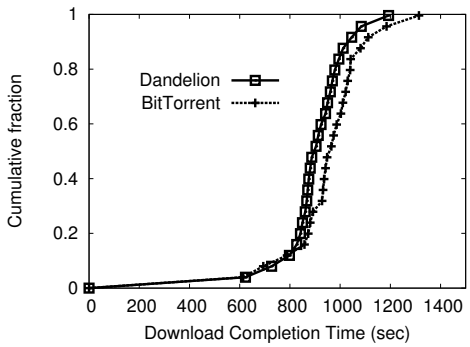

Figure 7: **CDF of download completion times of ∼ 400 BitTorrent and Dandelion clients that download a 100MB file. Dandelion and BitTorrent clients have average download completion time equal to 893 sec and 937 sec, respectively.**

a 100MB file divided in 256KB chunks. The duration of each experiment is 2200 sec. Free-riders never upload, nor do they expend credit. Cooperative clients always respond to chunk requests from their selected downloaders.

In each experiment, the swarm includes a group of 20 free-riders and a group of 20 cooperative clients all of which have upload and download rate-limits equal to 100KB/sec and 350KB/sec, respectively. In the rest of this evaluation we call the groups of the 20 free-rider and 20 cooperative clients, the *free-rider* and the *reference cooperative* group, respectively. The rest of the leechers are rate-limited according to the distribution used in § VI-B. Unless mentioned otherwise, cooperative and free-rider clients connect to roughly 50 and 350 peers at a time, respectively.

For each configuration we repeat the experiment 10 times and we extract mean values and 95% confidence intervals of client download rates. If the client completes its download during the experiment (not always true for free-riders), its download rate is equal to the size of downloaded content divided by the download completion time. Otherwise, its download rate is the size of downloaded content divided by the experiment duration.

Figure 8(a) compares the two groups when the portion of leechers that remain online to seed varies from 0 to 100%. With this measurement we show that the "large view" exploit (§ II-B) enables free-riders to tap into scarce system resources and harm compliant clients by monopolizing the seeders and exploiting optimistic unchoking. For comparison purposes, for each percentage of leechers that become seeders we also depict the download rate of the cooperative group in the absence of free-riders.

We observe that free-riders obtain almost equal download rates with their cooperative counterparts in under-provisioned swarms with 0% to 25% seeders. Compliant clients suffer a performance hit of approximately 15%, comparing to their performance in the absence of free-riders. When the swarm has 50% to 100% leechers that become seeders, free-rider clients achieve 5% to 10% higher download rates than cooperative ones. This result confirms the potential for wide adoption of free-riding. In well-provisioned swarms, the download rate of cooperative clients degrades by roughly 10% comparing to

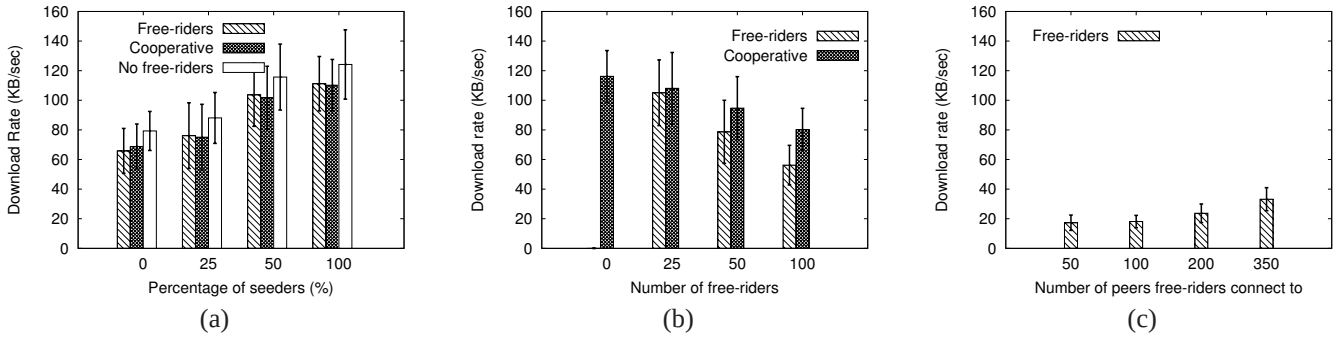

Figure 8: **Swarm-wide mean download rates of a group of 20 free-riders and a group of 20 cooperative clients that join a swarm of ∼350 leechers to download a 100MB file. (a) Download rates for varying percentage of peers that remain online seeding after download completion. We also depict the download rate of cooperative clients in the absence of free-riders ("No free-riders"); (b) Download rates for varying number of free-riders; (c) Download rates of free-riders when they cannot download from seeders, for varying number of peers that they connect to.**

their rate in the absence of free-riders.

In Figure 8(b), we compare the average performance of the free-rider and the 20-client reference cooperative group as the number of free-riders ranges from 0 to 100 clients. All leechers become seeders upon download completion. This measurement shows that the wide adoption of free-rider clients causes substantial performance degradation in BitTorrent swarms. When the number of free-riders varies in 50 to 100, the reference cooperative group attains approximately 20% to 30% worse performance than in a swarm with no free-riders. We also observe that as the number of free-riders increases, free-riders do not fare as well comparing to compliant clients. When there are 50 free-riders, free-riders have 20% worse performance than their cooperative counterparts.

Figure 8(c) depicts the performance of the free-rider group when free-riders do not download from seeders and cooperative clients employ a chunk-level TFT scheme. Under this scheme, leechers upload plaintext chunks to their selected downloaders as long as the trade surplus does not exceed 1. BitTorrent does not currently prevent free-riders from downloading from seeders. On the other hand, Dandelion seeders are motivated to upload only encrypted content, for which they are rewarded. No leechers become seeders. The number of peers to which free-riders connect to varies in 50 to 350 to illustrate the impact of the "large view" exploit.

As can be seen in Figure 8(c), when free-riders connect to 350 peers, they can attain up to 33KB/sec. Although this is not a good download rate by itself, recall that any gains of a Dandelion free-rider translate to monetary losses for its peers. With this measurement, we show that the credit-based exchange substantially reduces the free-rider download rates. We also motivate Dandelion's TFT-based exchange. We show that even if we employ credit-based exchange and enforce strict chunk-level tit-for-tat, free-riders that employ the "large view" exploit are able to download non-negligible amounts of content without expending credit or uplink bandwidth.

We note here that under Dandelion, free-riders may expend credit to download content and upload in exchange for credit, while they use "large view" to exploit the initial plaintext offers of their peers. In this way, free-riders achieve good download rates while saving on credit.

Our results show that the large view exploit is very effective under a flashcrowd if the client is able to connect to many hundreds of peers, especially if the swarm is well-provisioned with seeders. This indicates that BitTorrent not only lacks incentives for seeding but its rate-based tit-for-tat incentives are also manipulable. Our results also indicate that the exploit affects the performance of compliant clients, regardless of whether the swarm is well-provisioned; in all cases compliant clients incur performance degradation in the presence of free-riders. In addition, we demonstrate that without the cryptographic TFT-based exchange, the large view exploit allows Dandelion free-riders to download a substantial amount of content without expending resources.

VII. Related Work

In this section we discuss previous work on incentives for cooperation in peer-to-peer content distribution systems as well as previous work on cryptographic fair exchange.

A. Pairwise Currency as Incentives

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

An eMule [17] client rewards cooperative peers by reducing the time the peers have to wait until they are served by the client. Swift [52] introduces a pairwise credit-based trading mechanism for peer-to-peer file sharing networks and examines the available peer strategies. In [7], the authors suggest tackling free-riding in BitTorrent by employing chunk-level tit-for-tat, which is similar to pairwise credit incentives. Keidar et al. [53] present the design of a P2P multicast protocol in which a client tracks the difference between the amount of data the peer has sent to the client so far and the expected per-link throughput. They formally prove that their scheme fosters cooperation among selfish peers. 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 [54] combines pairwise credit balances with a transitive trading mechanism. OneHop [50] employs a one hop transitive reputation mechanism to incent cooperation in BitTorrent. These incentive mechanisms are based on the premise that a client remains perpetually interested in exchanging his earned credit or reputation for content downloads from the same network. Unlike Scrivener or OneHop, credit earned by Dandelion clients can be converted into monetary rewards, providing strong and immediate incentives for clients to upload, even if the network ceases to offer content that interests the client.

BAR Gossip [13] is suitable for P2P streaming of live content. Owing to its public-key-based cryptographic fair exchange mechanism it is robust to clients that attempt to free-ride. However, clients that receive initial optimistic offers from their peers need to expend bandwidth in order to reciprocate with invalid or old and irrelevant chunk transmissions. Its verifiable peer selection prevents clients from selecting many and specific victim peers to DoS attack. However, its verifiable peer selection technique assumes that no client can join the network after the streaming session starts. A consequence of verifiable peer selection is that BAR Gossip is resilient to the large view exploit. 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 incentivize seeding for static content distribution or video on demand, guarantees fair exchange of content uploads for virtual currency enabling two peers to trade even if they are not mutually interested in each other's content. In this work, we describe the specifics of combining Dandelion's credit-based exchange with BAR Gossip's balanced-exchange using trade surplus limiting and downloader selection.

B. 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 [55] employs a global virtual currency bank and certified-mail-based [56] 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. Furthermore, Karma relies on a secure DHT to ensure that credit queries are resolved by appropriate nodes. In contrast, Dandelion's centrally maintained global currency is non-manipulable by clients, enabling a commercial content provider to incentivize client cooperation by offering monetary rewards. Furthermore, Dandelion employs real currency circumventing the monetary regulation issues that Karma needs to address.

Horne et al. [57] 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. [38] 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 cryptographic fair exchange of chunk uploads for credit, effectively preventing a client from obtaining any chunks from selfish peers without having sufficient credit.

PPay [58], WhoPay [59] and more recently [43] are micro-payment proposals that employ public key cryptography and are designed for the P2P content distribution case. WhoPay has a distributed double-spending detection system based on a distributed DHT-based database, but it is vulnerable to peer collusions, and routing attacks on the DHT. MojoNation [60] 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. PeerImpact [46] provides monetary rewards for cooperative behavior. However, the exact mechanism with which the system exchanges service for credit is not publicly available and it appears not to offer our cryptographic non-manipulability guarantees.

A very recent BitTorrent extension [61] exchanges cryptographically signed proofs of service for content uploads. The above schemes do not guarantee fair exchange of content for payment. Free-riders may establish short-lived sessions to many peers, and download small portions of content or obtain payments from without paying or uploading. In addition, free-riders may send to the uploaders proofs of payment that do not reflect real credit value. As a result, free-riders may acquire substantial amount of content without uploading or paying credit, respectively. In addition, free-riders may send payments that do not reflect real credit value, also known as double-spending. WhoPay employs a decentralized double-spending detection mechanism that is based on a distributed DHT-based database, but this mechanism is vulnerable to peer collusions, and attacks on DHT routing.

Similar to Kazaa, Maze [62] users are rewarded points for uploading, and expend points for successful downloads.

Users that obtain more points than a specified threshold are assigned high bandwidth quotas. However, the system relies on users faithfully reporting the amount of content they exchange and it does not guarantee strong identities. Thus, as reported in [62], it is vulnerable to Sybil attacks. Furthermore, since uploads result in more credit gains than downloads result in charges, the system is susceptible to peer collusions. It is also vulnerable to source code modifications similar to the ones experienced in Kazaa [19]. Furthermore, credit in Maze does not correspond to real monetary value. Therefore, it does not incent peers that are no longer interested in downloading content from the network and special care must be taken to prevent starvation, inflation and deflation issues.

In Sharp [63] peers exchange signed tickets for basic resources such as computing, storage and network capacity. System participants can issue, subdivide, trade and use the tickets to allocate resources. However, it does not provide a reliable complaint resolution mechanism in case clients refuse to honor valid tickets for their resources. In contrast, because Dandelion is content-resource-oriented, the server is always able to resolve complaints by verifying the validity of the transferred content.

C. Reputation Mechanisms

Reputation mechanisms, e.g. [25, 64, 65], may allow seeders to rank their leechers based on the rate with which the leechers upload to their peers. By employing these mechanisms, the system can in theory prevent free-riders from downloading from seeders. As proposed in [66], the peers of leechers would report to the seeders, with which the leechers are connected, information about the rates with which the leechers uploads to their peers, and the reputation mechanism would be used to rank the truthfulness of the peer reports and the cooperativeness of the leechers. However, reputation systems are vulnerable to the Sybil attack [32] and collisions, especially in swarms with small populations, and in the best case offer probabilistic guarantees of reputation correctness.

Furthermore, reputation-based incentive mechanisms offer coarse-grained evaluation of a peer's level of cooperation, thus they are unsuitable for schemes that employ monetary rewards.

D. Double-spending Prevention

Osipkov et al. [67] propose a scheme to prevent double-spending in a micropayment-based market, under which clients purchase service from web servers. The following issues set obstacles in deploying their solution in the P2P content distribution setting. First, it requires a relatively static set of entities (e.g. peers) that act as witnesses/trackers of coin transactions, thus it is not suitable for a highly dynamic P2P environment. Second, this solution addresses witnesses that wrongfully claim that a coin has been double-spent, but in our setting this is not a compelling problem. We are mostly concerned with witnesses that collude with peers by lying that a coin has not been double-spent. Their solution assumes that witnesses don't have incentives to collude with clients under their web-server/client setting, but this assumption does not hold under our setting. Last, their scheme employs complex cryptography, making its correct implementation a difficult task.

E. 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 [68–72]. 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 [14, 15, 73, 74] or semi-trusted [75, 76] third party (TTP). The main differences of these solutions with our scheme are as follows: 1) In the optimistic fair-exchange schemes proposed in [14, 15, 73] the TTP is involved only when a party does not complete the transaction to carry out the transaction itself or issue affidavits on what happened during the transaction. This mechanism can be combined with micropayments to ensure the fair exchange of content for credit. However, unlike Dandelion's cryptographic protocol for credit-based exchange, such scheme would not prevent double-spending. Although the schemes in [14, 15] can determine whether a message originates from a party and whether it is the message that the originator initially intended to send, they cannot determine whether the message itself is valid, i.e. the integrity of a chunk. Our TFT-based exchange employs a public-key-based optimistic fair-exchange scheme similar to [15]. Our scheme however, similar to BAR Gossip, can determine whether the transmitted chunks are valid, uniquely identifying a dishonest peer;

2) Unlike [75], [77] and [76], our scheme does not rely on untrusted clients to become semi-TTP; 3) Unlike [74], our credit-based exchange 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. 4) Unlike in [14] and [77]'s setting, in Dandelion the transfer of the encrypted chunk itself is the expensive resource being exchanged. In case the sender misbehaves, the server would have to send the decrypted chunk himself. The server's resources would be exhausted in case senders or receivers misbehaved.One approach could be to distribute the TTP tasks among the peers, similar to how TRICERT [77] distributes the tasks among "postal agents". However, we cannot assume that the Dandelion peers can act as semi trusted third parties, because they would not have incentive to perform the expensive TTP task of uploading the decrypted chunk in case of complaints.

VIII. Conclusion

This paper describes Dandelion: an incentive scheme for cooperative (P2P) distribution of paid content. Its primary function is to enable a content provider to motivate its clients to contribute their uplink bandwidth.

Our scheme rewards cooperative clients with credit or with reciprocal uploads from their peers. Since it employs non-manipulable cryptographic schemes for the fair exchange of resources, the content provider can redeem a client's credit for monetary rewards. Thus, our design provides strong incentives for clients to seed content and eliminates free-riding.

Our experimental results 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 seeding substantially improves swarm-wide performance and that a Dandelion-based content distribution system can attain performance comparable to BitTorrent. It also demonstrates that the proposed hybrid incentive scheme significantly reduces the load on the server when compared to our previously fully centralized incentives. These facts illustrate the plausibility of our design choice: centralizing the incentive mechanism in order to increase resource availability in P2P content distribution.

Acknowledgments

We are thankful to Eddie Kohler, Nikitas Liogkas and the anonymous reviewers for their fruitful feedback on this work. This work was supported by NSF award CNS-0627166.

References

[1] "Hulu - Watch your Favorites. Anytime. For Free," www.hulu.com.
[2] "Music denied - Shoppers overwhelm iTunes," edition.cnn.com/2006/TECH/internet/12/28/itunes.slowdown.ap/index.html?eref=rss_topstories, December 2006.
[3] B. Cohen, "Incentives Build Robustness in BitTorrent," in *P2P Econ*, June 2003.
[4] C. Huang, J. Li, and K. W. Ross, "Can Internet Video-on-Demand Be Profitable?" in *SIGCOMM*, August 2007.
[5] "BitTorrent, Inc Launches The BitTorrent Entertainment Network," www.bittorrent.com/about/press/bittorrent-inc-launches-the-bittorrent-entertainment-network, Feb 2007.
[6] D. Hughes, G. Coulson, and J. Walkerdine, "Free Riding on Gnutella Revisited: The Bell Tolls?" in *IEEE Distributed Systems Online*, vol. 6, no. 6, June 2005.
[7] S. Jun and M. Ahamad, "Incentives in BitTorrent Induce Free Riding," in *P2P Econ*, August 2005.
[8] N. Liogkas, R. Nelson, E. Kohler, and L. Zhang, "Exploiting BitTorrent For Fun (But Not Profit)," in *IPTPS*, February 2006.
[9] M. Sirivianos, J. H. Park, R. Chen, and X. Yang, "Free-riding in BitTorrent Networks with the Large View Exploit," in *IPTPS*, February 2006.
[10] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer, "Free Riding in BitTorrent is Cheap," in *HotNets*, November 2006.
[11] M. Sirivianos, J. H. Park, X. Yang, and S. Jarecki, "Dandelion: Cooperative Content Distribution with Robust Incentives," in *USENIX*, June 2007.
[12] B. Wilcox-O'Hearn, "Experiences Deploying a Large-Scale Emergent Network," in *IPTPS*, March 2002.
[13] H. Li, A. Clement, E. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin, "BAR Gossip," in *OSDI*, November 2006.
[14] N. Asokan, M. Schunter, and M. Waidner, "Optimistic Protocols for Fair Exchange," in *ACM CCS*, April 1997.
[15] J. Zhou and D. Gollmann, "An Efficient Non-repudiation Protocol," in *CSFW*, March 1996.
[16] 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 Comput. Commun. Rev.*, vol. 33, no. 3, 2003, pp. 3–12.
[17] "The eMule Project," www.emule-project.net.
[18] N. Andrade, M. Mowbray, A. Lima, G. Wagner, and M. Ripeanu, "Influences on Cooperation in Bittorrent Communities," in *P2P Econ*, August 2005.
[19] "Kazaa Lite," en.wikipedia.org/wiki/Kazaa_Lite.
[20] "NRPG RatioMaster: Fake Upload and Download Stats of a Torrent to Almost all Bittorrent Trackers," www.brothersoft.com/nrpg-ratiomaster-78031.html.
[21] Q. Lian, Z. Zhang, M. Yang, B. Y. Zhao, Y. Dai, and X. Li, "An Empirical Study of Collusion Behavior in the Maze P2P File-Sharing System," in *ICDCS*, June 2007.
[22] M. Piatek, T. Isdal, A. Venkataramani, and T. Anderson, "One Hop Reputations for File Sharing Workloads," in *NSDI*, April 2008.
[23] M. Izal, G. Urvoy-Keller, E. Biersack, P. Felber, A. A. Hamra, and L. Garces-Erice, "Dissecting BitTorrent: Five Months in a Torrents Lifetime," in *PAM*, April 2004.
[24] Manaf and K. G. Anagnostakis, "On the impact of p2p incentive mechanisms on user behavior," in *NetEcon+IBC*, 2007.
[25] Q. Lian, P. Yu, M. Yang, Z. Zhang, Y. Dai, and X. Li, "Robust Incentives via Multi-level Tit-for-tat," in *IPTPS*, 2006.
[26] B. Fan, D.-M. Chiu, and J. C. Lui, "The Delicate Tradeoffs in BitTorrent-like File Sharing Protocol Design," in *ICNP*, November 2006.
[27] M. Bellare, R. Canetti, and H. Krawczyk, "Keying Hash Functions for Message Authentication," in *LNCS*, vol. 1109, 1996.
[28] "The Transport Layer Security (TLS) Protocol, Version 1.1," http://www.ietf.org/rfc/rfc4346.txt.
[29] "Quote from PACIFIC BELL: $18000 per month for an OC3 line," shopforoc3.com/, Mar. 2006.
[30] J. Gray, "Distributed Computing Economics," Microsoft Research, Tech. Rep., 2003, MSR-TR-2003-24.
[31] J. Shneidman, C. Ng, D. C. Parkes, A. AuYoung, A. C. Snoeren, A. Vahdat, and B. N. Chun, "Why Markets Could (But Don't Currently) Solve Resource Allocation Problems in Systems," in *HotOS-X*, June 2005.
[32] J. R. Douceur, "The Sybil Attack," in *IPTPS*, March 2002.
[33] J. Steiner, C. Neuman, and J. Schiller, "Kerberos: An Authentication Service for Open Network Systems," in *USENIX Winter Conference*, 1988.
[34] B. Cheng, X. Liu, Z. Zhang, and H. Jin, "A measurement study of a peer-to-peer video-on-demand system," in *IPTPS*, 2007.
[35] A. Bharambe, C. Herley, and V. Padmanabhan, "Analyzing and Improving a BitTorrent Networks Performance Mechanisms," 2006.
[36] A. Legout, N. Liogkas, E. Kohler, and L. Zhang, "Clustering and Sharing Incentives in BitTorrent Systems," June 2007.
[37] A.-T. Gai, F. Mathieu, F. de Montgolfier, and J. Reynier, "Stratification in P2P Networks: Application to BitTorrent," in *ICDCS*, 2007.
[38] J. Li and X. Kang, "Proof of Service in a Hybrid P2P Environment," in *ISPA Workshops*, 2005.
[39] "Broadband promotions," http://www.broadband-promotions.net/.
[40] "Verizon FiOS Internet Packages and Prices," http://www22.verizon.com/content/consumerfios/.
[41] "Shop for bandwidth," www.shopforbandwidth.com/t1-lines-t3-lines-ds3-oc3-faste-gige-service.php.
[42] "Amazon simple storage service." [Online]. Available: \url{http://www.amazon.com/gp/browse.html?node=16427261}
[43] M. J. Freedman, C. Aperjis, and R. Johari, "Prices are Right: Managing Resources and Incentives in Peer-assisted Content distribution," in *IPTPS*, February 2008.
[44] D. C. Parkes, R. Cavallo, N. Elprin, A. Juda, S. Lahaie, B. Lubin, L. Michael, J. Shneidman, and H. Sultan, "ICE: An Iterative Combinatorial Exchange," in *ACM Conference on Electronic Commerce*, 2005.
[45] "ROO Online Video Network," http://www.roo.com/.
[46] "Peer impact," http://en.wikipedia.org/wiki/Peer_Impact.
[47] "RedSwoosh, an Akamai Company," http://www.akamai.com/redswoosh.
[48] B. S. Frey, , and R. Jegen, "Motivation crowding theory," pp. 589–611, 2001.
[49] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat, "Maintaining High Bandwidth Under Dynamic Network Conditions," in *USENIX*, April 2005.
[50] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani, "Do Incentives Build Robustness in BitTorrent?" in *NSDI*, April 2007.
[51] M. Dischinger, A. Haeberlen, K. P. Gummadi, and S. Saroiu, "Characterizing residential broadband networks," in *IMC*, October 2007.
[52] K. Tamilmani, V. Pai, and A. Mohr, "SWIFT: A System with Incentives for Trading," in *P2P Econ*, August 2004.
[53] I. Keidar, R. Melamed, and A. Orda, "EquiCast: Scalable Multicast with Selfish Users," in *PODC*, July 2006.
[54] P. Druschel, A. Nandi, T.-W. J. Ngan, A. Singh, and D. Wallach, "Scrivener: Providing Incentives in Cooperative Content Distribution Systems," in *Middleware*, 2005.

[55] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer, "Karma: A Secure Economic Framework for P2P Resource Sharing," in *P2P Econ*, June 2003.
[56] B. Schneier, Applied Cryptography, 2nd edition, 1995.
[57] B. Horne, B. Pinkas, and T. Sander, "Escrow Services and Incentives in Peer-to-peer networks," in *EC*, 2001.
[58] B. Yang and H. Garcia-Molina, "PPay: Micropayments for Peer-to-peer Systems," in *ACM CCS*, October 2003.
[59] 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*, June 2006.
[60] "The MNet Project," mnetproject.org.
[61] "The Snowball Traceability Protocol," http://developer.snowballnetworks.com/trac.
[62] M. Yang, H. Chen, B. Y. Zhao, Y. Dai, , and Z. Zhang, "Deployment of a Large-scale Peer-to-Peer Social Network," in *WORLDS*, 2004.
[63] Yun Fu and Jeffrey Chase and Brent Chun and Stephen Schwab and Amin Vahdat, "Sharp: an architecture for secure resource peering," in *SOSP*, 2003.
[64] S. D. Kamvar, M. Schlosser, and H. Garcia-Molina, "The EigenTrust Algorithm for Reputation Management in P2P Networks," in *WWW*, 2003.
[65] A. Blanc, Y.-K. Liu, and A. Vahdat, "Designing Incentives for Peer-to-Peer Routing," in *INFOCOM*, 2004.
[66] M. Li, J. Lu, and J. Wu, "Free-riding on bittorrent-like peer-to-peer file sharing systems: Modeling analysis and improvement," in *TPDS*, 2007.
[67] I. Osipkov, E. Y. Vasserman, N. Hopper, and Y. Kim, "Combating double-spending using cooperative p2p systems," in *ICDCS*, 2007.
[68] E. F. Brickell, D. Chaum, I. Damg, and J. V. de Graaf, "Gradual and Verifiable Release of a Secret," in *CRYPTO*, 1988.
[69] R. Cleve, "Controlled Gradual Disclosure schemes for Random bits and their Applications," in *CRYPTO*, 1989.
[70] I. B. Damg, "Practical and Provably Secure Release of a Secret and Exchange of Signatures," in *EUROCRYPT*, 1994.
[71] T. Okamoto and K. Ohta, "How to Simultaneously Exchange Secrets by General Assumptions," in *CCS*, 1994.
[72] S. Even, O. Goldreich, and A. Lempel, "A Randomized Protocol for Signing Contracts," in *Communications of the ACM*, vol. 28, no. 6, 1985.
[73] F. Bao, R. Deng, and W. Mao, "Efficient and Practical Fair Exchange Protocols with Off-line TTP," in *S&P*, 1998.
[74] J. Zhou and D. Gollmann, "A Fair Non-repudiation Protocol," in *IEEE S&P*, 1996.
[75] K. Franklin and M. K. Reiter, "Fair exchange with a semi-trusted third party," in *CCS*, 1997.
[76] M. K. Franklin and G. Tsudik, "Secure Group Barter: Multi-party Fair Exchange with Semi-Trusted Neutral Parties," in *Financial Cryptography*, 1998.
[77] G. Ateniese, B. de Medeiros, and M. T. Goodrich, "TRICERT: A distributed certified E-mail scheme," in *NDSS*, 2001.

Michael Sirivianos is a Ph.D. candidate in Computer Science at the University of California, Irvine. His research interests include cooperative content distribution and human verifiable secure device pairing. He received a B.S in Electrical and Computer Engineering from the National Technical University of Athens in 2002, and an M.S. in Computer Science from the University of California, San Diego in 2004.

Xiaowei Yang is an Assistant Professor of Computer Science at the University of California, Irvine. Her research interests include congestion control, quality of service, Internet routing architecture, and network security. She received a B.E. in Electronic Engineering from Tsinghua University in 1996, and a Ph.D. in Computer Science from MIT in 2004.

Stanislaw Jarecki is an Assistant Professor of Computer Science at the University of California, Irvine. His research interests are cryptography, security, and distributed algorithms. He received a B.S. in Computer Science from MIT in 1996, and a Ph.D. in computer science from MIT in 2001.