Floodgate: A Micropayment Incentivized P2P Content Delivery Network

Srijith K. Nair*, Erik Zentveld*, Bruno Crispo*†, Andrew S. Tanenbaum*
*Department of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
†University of Trento, Italy
{srijith,fmzentve,crispo,ast}@cs.vu.nl

***Abstract*—As the sale of digital content is moving more and more online, the content providers are beginning to realize that bandwidth infrastructures are not easily scalable. The emergence of peer-to-peer content delivery networks present these providers with a way to overcome this limitation. However, such networks have so far been ad-hoc in nature. One of the main reason for this has been the lack of incentives for end users to contribute their bandwidth to the network. In this paper we present the design and implementation of a peer-to-peer protocol named Floodgate that provides a micropayment based incentive for peers to contribute their bandwidth. Floodgate implements an optimistic fair exchange protocol and is designed to be resilient against targeted attacks. Performance measurements, including those conducted over the PlanetLab infrastructure, show that Floodgate's security and cryptographic overheads are low when compared against the standard BitTorrent implementation.**

I. Introduction

As the entertainment industry is becoming more and more digitized, the traditional delivery network used to advertise, commercialize and deliver its products is rapidly migrating over to the Internet. A very large number of companies, from iTunes to Amazon, have signed agreements with content providers (i.e. music companies) to sell their products online. These new revenue opportunities however do not come for free to the reseller. In order to distribute digital content to customers over Internet, companies like Amazon have to make sure to always have enough bandwidth to serve the content with an acceptable quality of service. So far, this issue has been addressed in two ways, by using expensive third-party content delivery networks (CDNs) like Akamai, to reduce latency, and by buying enough bandwidth to serve the content directly with their own infrastructure. Both solutions are quite expensive, because bandwidth is usually sold based on the peak required (the maximum bandwidth required during the day) rather than the bandwidth effectively used.

This situation has motivated some companies to explore alternative means of implementing their delivery infrastructure and in particular the use of peer-to-peer (P2P) technology for this purpose. Recent effort by content providers to use BitTorrent Inc.'s infrastructure [2] is a step in this direction. However, even such solutions do not exploit the true scalability provided by the P2P architecture since they do not involve the clients and do not utilize the idle bandwidth available at these end users.

This work has been supported by NWO project ACCOUNT 612.060.319 and partially by the EU FP6 project GridTrust contract 0033817.

One of the hurdles in the path towards harnessing these clients' bandwidth is the absence of a good incentive model to motivate them to contribute their bandwidth towards the network. Rational users tend not to contribute their resources in the absence of such incentive models. The tit-for-tat mechanism [3] used by traditional P2P systems like BitTorrent does not work in this case since all it provides is an incentive for the clients to upload while still downloading, i.e. it does not incentivize the seeding (uploading when not downloading) of the content. Furthermore, as shown by others [4], [5], the mechanism is susceptible to free-rider attacks that allow the attacker to download much more than the uploaded amount.

In this paper we present the design and implementation of Floodgate, a P2P system that provides monetary incentive to the clients to contribute their bandwidth to the system. Floodgate is able to provide a robust, fair system by exploiting the closed, self-contained nature of the provider's network. This micropayment architecture incentivize the peers to be part of the network on a long term basis, allowing for the formation of a stable and scalable CDN.

The rest of the paper is organized as follows. In Section II we describe the popular BitTorrent P2P protocol in brief. We present the details of Floodgate in Section III and consider various attacks scenarios in Section IV. In Section V we present the results of the performance measurements made using an implementation of Floodgate. In Section VI we discuss how Floodgate's approach is different from other works in this area. We conclude in Section VII.

II. BitTorrent

BitTorrent is a P2P protocol that uses tit-for-tat mechanism [3] to incentivize peers to upload data in order to download more of it. The file to be shared on a BitTorrent network is split into fixed-size *pieces*, all of which, except the last, are of equal length. A SHA-1 hash is computed over each piece in order to verify the integrity of these pieces. The hashes are published in a meta file called a *torrent* file which also contains details of the file like its total length, name and details of the *trackers* that needs to be contacted by the peer to get information about other peers from whom the pieces can be downloaded.

978-1-4244-2390-3/08/$25.00 ©2008 IEEE

After downloading the torrent file, the peer sends HTTP requests to the tracker specified in the torrent, with details about itself, like the file it is interested in, the port it is listening to and index of pieces it has already downloaded (if any). The tracker replies back with a list containing the contact information of peers which are in various stages of downloading pieces of the same file. This list is randomly generated and is usually a subset of the entire list of peers the tracker knows about.

The client is responsible for contacting these peers and requesting pieces from them. A peer typically maintains about 40 connections with others peers, called the *neighbor set*. Not all of these connections are used for downloading at the same time. The extra connections are used to keep track of the state of the pieces in the swarm – which pieces are common and which are rare. This information could then be used to decide which piece to ask for next.

Peers communicate with each other, specifying which pieces they have and which pieces they want. When contacted, a peer uses a *choking* algorithm to determine whether to allow the requesting peer to download the piece. In the choking algorithm, connection endpoints have the following attributes (a) choked/unchoked: refusal or permission to download (b) interested/uninterested: peer wants piece from the other side. New connections start out uninterested and choked. The advertisement of available pieces can be used to determine the interest/disinterest in a piece. Periodically each peer selects a list of *preferred peers* which are interested in a piece it has, based on the download rate of the peers and uploads to only these fastest peers. If a peer has the complete file, it uses the other peer's upload rate rather than download rate to decide which peers to unchoke.

To make sure new peers that are joining can get pieces to trade, optimistic unchoking is used. In *optimistic unchoking*, at any one time there is a single peer which is unchoked regardless of its upload rate (if interested, it counts as one of the allowed downloaders). Which peer is optimistically unchoked rotates every 30 seconds. Newly connected peers are three times as likely to start as the current optimistically unchoked node as anywhere else in the rotation. This gives them a decent chance of downloading a complete piece to then upload. For the existing members of the swarm, optimistic unchoking can be used as a way to discover new peers that may possibly have a better upload rate.

III. FLOODGATE

Floodgate, the P2P scheme presented in this paper, proposes the use of micropayments incentivized clients to contribute their bandwidth to the network. The protocol is designed to not only provide optimistic fairness guarantees but also to be resilient against free-riding and other targeted attacks.

For the ease of understanding, in this paper we use terminologies similar to those used in BitTorrent protocol to describe Floodgate's working.

Before we present the details of the protocol, it is beneficial to understand the application context of Floodgate. The protocol is aimed at a content provider networks with the following properties: (1) the network is managed by a central entity $\mathbb{T}$ (for example like Amazon) (2) each customer (peer) wishing to join the network registers and creates an account with $\mathbb{T}$, the account being tied to a real-world monetary account like credit card or bank account

A. Protocol Overview

First we provide a brief overview of Floodgate's architecture before diving into the details of the protocol steps.

The content provider $\mathbb{T}$ controls the tracker as well as the web server that serves the torrent file. Unlike in BitTorrent, the tracker plays a more involved role in Floodgate, as explained below.

Once a peer c_1 registers itself with $\mathbb{T}$ and creates an account, when it wishes to download a content, it searches for the associated torrent file in a known public repository operated by $\mathbb{T}$ and downloads it. c_1 then contacts the tracker specified in the torrent file. The tracker replies with the list of peers in the network who are in various stages of downloading or seeding the content. In addition, it also sends to c_1 micropayment tokens that can be used by the peer to pay for content pieces it successfully downloads from various peers in the network. Just as in BitTorrent, the peer then contacts other peers in the list given by the tracker to start downloading pieces of the content. After each successful download of a piece, c_1 sends the appropriate micropayment token to the peer from which it downloaded the piece. Each received token is redeemable at $\mathbb{T}$ in the form of future download credit or other payment options like cash, as the case may be. The exact business model adopted by $\mathbb{T}$ is independent of the working of Floodgate, making Floodgate more flexible.

While the steps mentioned above may look like minor modifications to BitTorrent protocol, the need to make the system secure, resilient to coordinated colluding attacks and at the same time efficient and scalable makes the design changes non-trivial in nature. In the rest of this section we look into details of the protocol steps.

B. Registration

When a new peer wishes to join the network, it registers itself with $\mathbb{T}$ after providing, among others, details like bank account or credit card number. It is then given a system wide unique user-ID uid. This uid is different from the BitTorrent $peerId$ which is unique only for a specific torrent/tracker. The peer then creates a public key-private key pair (EK_{c_1},DK_{c_1}) and sends the public key to $\mathbb{T}$. $\mathbb{T}$ sends back a signed certificate linking the public key of the peer with its uid. It is assumed that the public key of $\mathbb{T}$ is publicly known and certified by a commercial Certificate Authority, like VeriSign.

C. Payment token

In Floodgate, every content piece downloaded from a peer needs to be paid for by using a micropayment token. Some implementation of BitTorrent allows for downloading of content fragments smaller than a piece with the aim of increasing

the throughput of the download by engaging more peers in parallel. However since the integrity hash is computed over a piece (and not sub-pieces), if the hash of a piece does not match that specified in the torrent file, there is no way for the downloader to figure out which of the sub-pieces are the corrupt ones. Hence there is no way to identify which of the peers sent the (corrupt) sub-piece and should not be credited for the upload. So, in Floodgate, the smallest downloadable entity is fixed as a piece.

Mechanisms like hashchains [6] and asymmetric key based signatures, though secure, need computation intensive operations as well involved infrastructure setup, making them less attractive for use in generating the payment tokens. Instead, a very simple low overhead mechanism is used in Floodgate. A random number is generated for every piece that make up a content cid for every request, and this serves as the token for that piece. Thus, for every tracker request received, the tracker generates a specific number of random numbers for the couple $\langle uid, cid \rangle$. The use of random numbers for tokens enables them to be precalculated, allowing the tracker to scale well even when serving a large number of peers.

The tracker thus maintains a table like in Table I for each response it sends to the peer (rid is the request ID).

TABLE I
TABLE RECORD FOR EACH TRACKER REQUEST

rid	uid	cid	#pieces	token	token hashes
85	404	38	106	n_1^{85},n_2^{85},...n_{106}^{85}	h(n_1^{85}),...h(n_{106}^{85})
86	500	12	38	n_1^{86},n_2^{86},...n_{38}^{86}	h(n_1^{86}),...h(n_{38}^{86})
..	..	..	..	..	..
97	520	38	106	n_1^{97},n_2^{97},...n_{106}^{97}	h(n_1^{97}),...h(n_{106}^{97})

This list of tokens are sent to the requesting peer, signed using $\mathbb{T}$'s private key and encrypted with the peer's public key. Thus an attacker cannot decrypt and obtain the tokens. At the same time, the intended recipient peer can verify the authenticity of the tokens by verifying the signature of the token list.

$c_1 \rightarrow \mathbb{T}$: n_1, $DK_{c1}(TrackerReq)$
$\mathbb{T} \rightarrow c_1$: $n_1 + 1$, $DK_{\mathbb{T}}(peerlist)$, $EK_{c_1}(DK_{\mathbb{T}}(tokens))$

In the notation, n_1 is a nonce used to prevent replay attacks and signing is denoted as encryption using the private key. A signed message includes the content of the message as well as the signature on it. Note that the peer list can be sent without encryption because, unlike the tokens, it is not a secure resource.

D. Downloading pieces

Floodgate peer c_1 use the same mechanism as in BitTorrent to find out which peers have pieces which it is interested in obtaining. Once such a peer c_2 has been identified, c_1 sends a request to c_2. c_2 replies with the requested block, signed with the c_2's certified private key. On receiving the block, c_1 verifies the signature to ensure the identity of the peer that sent the block. It then sends the payment token corresponding

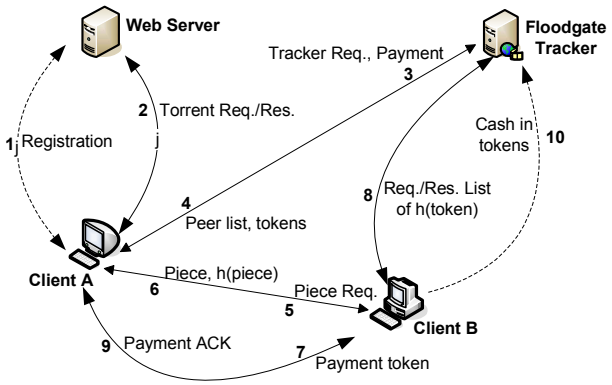

Fig. 1. Summary of Floodgate protocol steps

to the received block to c_2, encrypted with c_2's public key. If c_2 does not receive the payment token for the piece it uploaded, it puts c_1 in its internal blacklist and does not respond to any more requests for pieces from c_1.

$c_1 \rightarrow c_2$: n_2, EK_{c_1} , $\mathbb{T}cert_{c_1}$, $DK_{c_1}(PieceReq)$
$c_2 \rightarrow c_1$: n_2+1, n_3, EK_{c_2} , $\mathbb{T}cert_{c_2}$, Piece, $DK_{c_2}(h(piece))$
$c1 \rightarrow c2$: $n_2 + 2$, $n_3 + 1$, $EK_{c2}(DK_{c1}(token))$

E. Verifying payment

When c_2 receives the micropayment token from c_1, it contacts $\mathbb{T}$ to verify if the token is valid. In order to improve the protocol's efficiency, instead of sending the actual token to $\mathbb{T}$ and checking for a confirmation, c_2 asks for the hash of all the tokens handed out by $\mathbb{T}$ to c_1 for that specific content cid (i.e.$\langle uid, cid \rangle$). $\mathbb{T}$ replies with the relevant list of hashes. c_2 uses this to verify that the hash of the key sent by c_1 indeed matches the corresponding hash contained in the hash list sent by $\mathbb{T}$. Furthermore, the next time c_1 pays c_2 for another piece, c_2 can use the locally cached hash list to verify the payment and doesn't have to contact $\mathbb{T}$, speeding up the process.

$c_2 \rightarrow \mathbb{T}$: n_4, Hash Req. , cid , c_1
$\mathbb{T} \rightarrow c_2$: $n_4 + 1$, n_5, $DK_{\mathbb{T}}(h(n_1^{85}), h(n_2^{85})...h(n_{106}^{85}))$
$c_2 \rightarrow c_1$: $n_4 + 2$, $n_5 + 1$, $DK_{c_2}(PaymentOK)$

If the payment token received is not valid, c_2 asks c_1 to resend the token. Since the token cannot be spent multiple times, resending it is a safe operation. If, after repeated attempts, a valid token is not received, c_2 adds c_1 to its local blacklist and approaches $\mathbb{T}$ for mediation, providing it with $DK_{c_1}(PieceReq)$ and $EK_{c_2}(DK_{c_1}(token))$ as proof of transaction. While in our current implementation the mediation is performed manually, it can be automated without any changes to the protocol itself. Attacks stemming from the mediation capability is discussed in detail in Section IV-B.

F. Claiming payment

At periodic intervals, the peers contact $\mathbb{T}$ to encash the payment tokens it has accumulated till date. $\mathbb{T}$ will then

deduct the payment from one client and credit it to the recipient client. How exactly the token is credited may vary across systems. For example, some systems may give cash in exchange for the tokens while others may give store credit that can be used to pay for any downloads the peer may perform later.

$c_2 \rightarrow \mathbb{T}$: $EK_{\mathbb{T}}(DK_{c_2}(c_1|cid87|token5, c_3|cid45|token245))$
$\mathbb{T} \rightarrow c_2$: $EK_{c2}(DK_{\mathbb{T}}(OK|h(c_1|cid87|token5), ...$

Fig. 1 summarizes all the important steps involved in the Floodgate protocol explained above.

IV. Attacks

In this section we show how the various design choices in Floodgate enable it to withstand attacks targeted against it. We assume that the registered users of the systems may try to subvert the system by, among other things, colluding with other attackers and by trying to impersonate other users. We assume however that the attackers are not capable of breaking the crypto-primitives used in Floodgate, like public key cryptography, collision-resistant hashing etc.

A. Existing Attacks

Several attacks aimed at the tit-for-tat mechanism used by P2P systems like BitTorrent have surfaced over the years. Implementations like BitThief [7] aim to either download without any uploading, or maximizing the download speed while minimize the upload rate. Another attack strategy is to ask for pieces from seeders who have the complete file and hence would not request any pieces in exchange.

Since Floodgate does not rely on the ranked tit-for-tat mechanism to incentivize peers to share their bandwidth, these attacks are moot in our scheme. Instead, a selfish peer is actually encouraged to upload content because it gets paid for its contribution.

B. Free-rider Attack

Suppose c_2 sends a piece to c_1 on receiving a request but due to some network issue the piece is not received properly or the piece gets corrupted in transit. The hash check would fail at c_1's side and hence c_1 would refuse to pay for the piece. However c_2 may be under the impression that it had sent the piece as requested and, on not receiving a payment within a set time period, would put c_1 into its blacklist. In order to avoid this misunderstanding, c_1 should ask c_2 to resend the failed piece. As long as it is a resend request, c_2 should honor the request, since having more than one identical piece does not increase its value to c_1. Of course, in order to prevent a denial of service attack, c_1 and c_2 would set a maximum number of tries for such resend requests.

However, this altruistic behavior is prone to the following free-rider attack. Consider a content composed of 10 pieces. An attacker X would request 10 peers in the peer lists for each of the pieces. On receiving the pieces successfully, X would either just not pay up or reply to the respective peers that it had not received the piece correctly and keep requesting for the piece to be resent. After a threshold number of resends, these peers would stop responding to the resend requests. Either way, the peer would then add X to its blacklist and report X to $\mathbb{T}$ (sending the signed block request from X as proof). However, by then X has successfully downloaded the content completely and could safely disconnect.

The safest way to prevent this attack is to ensure that the number of pieces (N_p) that constitute the content is greater than the number of peers that X can contact (N_c), i.e. $N_p > N_c$. This can be done either by increasing the number of pieces in a content or by decreasing the number of peers in the peer list sent to the clients.

Increasing the number of pieces (by dividing the content into smaller sized piece chunks) has the additional benefit that the peer can increase the download rate by contacting larger number of peers for individual pieces. The downside is that doing so increases the size of data stored in the meta-info torrent file (SHA1 hash) as well as increases the number of tokens that needs to be generated for each instance of the content download.

The amount of peers in swarm can be limited in two ways (1) having a single swarm with membership limit of N_c and (2) dividing peers into separate distinct swarms, each with membership limit of N_c.

While the second option may seem the obvious choice, since it can support $N_c * N_s$ peers (where N_s is the number of distinct swarms), it is not secure against attack as the separation of swarms is an artificial one. If two attackers are able to position themselves in different swarms (say, by asking for the torrent at different points in time), they could exchange peer list among themselves to learn about members of the other's swarm and then contact these clients and defeat the security in place.

The combined strategy of increasing the number of pieces and setting a hard limit on the number of members in the swarm turns out to be the most efficient and secure defense against the attack. For example, consider a content of size 700MB split into 700 pieces of 1MB each. If we assume that the peer c_2 will respond to a request by peer c_1 only if c_1 has *no* outstanding payments due for pieces received before, the swarm can have a maximum membership of 701 members (N_c + 1).

Following this approach provides a very secure setup at the expense of the number of clients that can be served. An alternative approach that supports unlimited number of clients can be designed by leveraging the observation that X's payments details are accessible to $\mathbb{T}$ and hence there is always the possibility of paying the cheated peers as long as the cheating is recognized. On receiving consistent complaints against X from various member peers, $\mathbb{T}$ could identify X as an attacker and use the amount charged from X, to pay the cheated peers.

Identifying X as the attacker may not however be straightforward. For example, if we assume that $\mathbb{T}$ would brand X a cheat if it receives as may complaints against X as the

number of pieces in the content (conservative approach), X could thwart this by colluding with another peer which would serve X the piece but not complain to $\mathbb{T}$ on not receiving the payment. Practical solutions are however still possible. For example, a cost-risk analysis by $\mathbb{T}$ may deem that if n or more peers ($n > N_p$) complain about a particular X not paying up for a specific cid's pieces served, it is most probably not a denial of service attack against X and that X is indeed the attacker. n could either be decided as a fixed percentage of the number of pieces in the content or as a varying percentage based on, among other things, the value of the digital content.

C. Attack on token list

The list of payment token sent by $\mathbb{T}$ in response to a tracker request is encrypted with the public key of the peer which sent the signed request. This eliminates the possibility of an attacker X impersonating as peer c_1 as it cannot sign the request. Similarly the attacker cannot eavesdrop on the exchange and obtain the token list from the encrypted message.

D. Impersonating as uploader

Attacker X may try to launch a man-in-the-middle attack on the payment step by trying to get hold of payment sent by c_1 to c_2 after the piece has been received. However, this is prevented by c_1 sending the payment token encrypted with the certified key of the peer which sent the piece in the previous step. Note that when c_2 sends the piece to c_1 it also signs the piece. In our threat model we do not consider an attacker that is powerful enough to remove parts of messages from the networks, only those which can at most delay the receipt of these messages.

E. Invalid payment

An attacker could try and fool peer c_2 after receiving a piece from it by sending back an invalid token. At this point c_2 could be in one of two states. It could be the first time c_2 has sent a piece to x, in which case c_2 requests a hash of the payment token list from $\mathbb{T}$ for X and that particular content. On receiving the list of token hashes, c_2 would realize that hash of the (invalid) token sent by X does not match the hash sent by $\mathbb{T}$. c_2 would then blacklist X and forward the signed invalid token to $\mathbb{T}$ for further disciplinary action. If c_2 has already sent pieces to X successfully before, the list of token hashes would already be available locally and this can be checked to discovery the attempt at cheating. Note that the token sent by X is signed by it to prevent repudiation.

F. Guessing the token

An attacker could try and guess the micropayment token by brute force. This can be prevented by keeping the token space sufficiently large. Note that it is not enough for X to guess any token number from the random number space (which is fairly easy) but it should be a valid tuple $\langle uid|cid|token\rangle$. A fairly efficient pseudo-random number generator would be able to prevent such guesses. In addition, if a certain number of consecutive false claims have been received from client X, $\mathbb{T}$ could refuse to reply to claim requests for some time, in an effort to thwart the client's attempt to use $\mathbb{T}$ as a random oracle.

G. Wrongful ban attack

Attacker X could try to impersonate c_1 and ask for a piece from c_2 and then refuse to send the payment token in the hope that c_2 would report c_1 and get it blacklisted. However, since every piece request has to be signed with the private key of the peer, and since X cannot create c_1's signature, this impersonation attempt would fail.

Another related attack is to impersonate c_2 and advertise the availability of a piece that c_1 is interested in. However when c_1 requests the piece, X would send junk instead in the hope that c_1 will complain about c_2 and blacklist it. However, since every piece sent has to be accompanied by a signature on the hash of the piece, x cannot make it look like c_2 sent the junk data since X is not in possession of the private key of c_2 needed to create c_2's signature.

H. DoS against tracker

The tracker is a central component in the system and an attacker could try and launch a Denial of Service attack against it. General defense mechanisms [8] can mitigate this problem.

V. Results and Discussions

A Floodgate prototype was implemented using Python to measure its performance. This section reports the result of measurements conducted on the PlanetLab network [9]. The tracker server was a 2GHz, 1GB RAM, AMD64 3000+ machine run within the university network.

First, we measured the time taken to download a 10MB file with varying piece size in a network of one seeder and one leecher. The aim was to analyze the large cryptographic overhead introduced when small piece sizes are used.

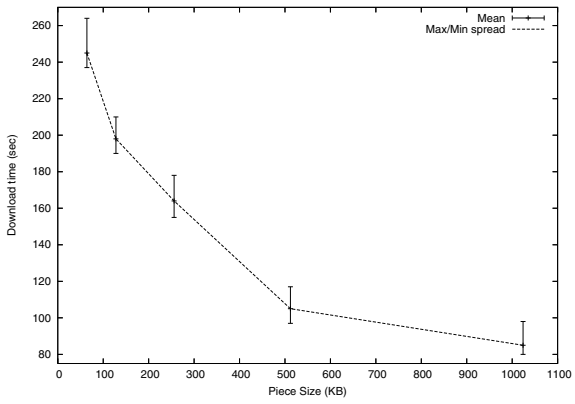

Fig. 2. Download time for varying piece sizes

As seen in Fig. 2, when the piece size is increased, the download time decreases. Since an increase in piece size means a decrease in the number of pieces that makes up the 10MB file, the number of times the encryption/decryption

TABLE II
DOWNLOAD TIME FOR VARYING OUTSTANDING UNREWARDED PIECES

Max. outstanding	Time (sec)
1	1128
2	1098
3	1023

TABLE III
DOWNLOAD TIME FOR FLOODGATE AND BITTORRENT FOR VARYING FILE SIZES

Size (MB)	Floodgate	BitTorrent	% difference
100	546	430	27%
250	819	635	29%
500	1140	880	29%

TABLE IV
AVG. CPU UTILIZATION FOR LEECHERS

Protocol	Utilization
Floodgate	72%
BitTorrent	61%

routines are executed decreases, allowing the seeder and leecher to serve and download the pieces faster.

The above experiment was performed assuming one outstanding piece limit i.e. after every piece that the uploader sends over to the leecher, it waits for the payment before replying to another request from the same client. In order to observe the effect of this limit on download time, experiments were performed for varying piece limits for a 700MB file, as tabulated in Table II. The very small variation in the download time shows that the time spent in waiting for the payment-ok message is insignificant as long as the uploader is able to perform a local verification using the list of token hashes fetched from the tracker. Note however that increasing the limit on unpaid-for pieces comes at the increased risk of free-rider cheating by the downloading peer.

The limit on the number of peers in the swarm is related to the number of pieces making up the content as $p > n*(s+d_p)$, where p is the number of pieces in the content, n is the number of outstanding unpaid pieces allowed per tuple, s is the number of seeders and d_p is the number of leechers who have at least one piece to upload. Thus, while there might be a small increase in the download rate, the number of allowed peers in the system will have to be reduced or the size of the pieces have to be decreased (in order to increase the number of pieces), both of which are detrimental to the overall performance of the network.

Next, the effect of multiple seeders were studied for various file sizes. The swarm starts off with 10 initial seeders. 100 leechers join in at about the same time to simulate a flash crowd. Once a leecher has downloaded the content completely, it stays behind to seed. Table III shows the average time taken for each leecher to complete the download of the various-sized content file. In order to compare Floodgate's performance with that of a normal BitTorrent implementation, the same experiments were performed with the BitTorrent Python reference client. The numbers show that Floodgate is about 30% slower than BitTorrent in downloading the files. This is mainly due to the large number of encryption, decryption and other additional cryptographic computations required by the Floodgate protocol. Our belief is that optimizing the Floodgate implementation would provide better performance, especially since the prototype implementation had to resort to mix-and-match approach to choosing cryptographic libraries to satisfy various functional requirements.

In the next set of experiments, we looked at the CPU utilization of a system when Floodgate was running and compared it against that running a BitTorrent client. In order to obtain a real estimate, the experiments were conducted not on a virtual machine based PlanetLab machine but on a desktop running Ubuntu Linux 2.6.15-28 with 1.5GHz CPU and 512MB memory. Table IV presents the finding. As expected, Floodgate client consumes more CPU than BitTorrent due to the additional cryptographic operations involved. A similar increase in CPU utilization is seen for the seeding peer too. In addition, it was also seen that increasing the piece size decreased the CPU utilization since the number of calls to the heavy cryptographic routines decreased due to smaller number of pieces constituting the content, as shown in Fig. 3.

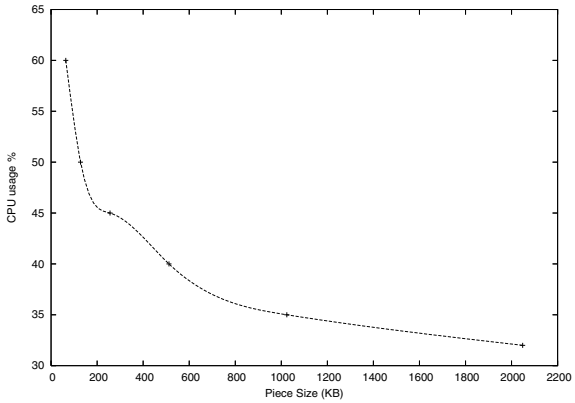

Fig. 3. CPU usage for varying piece sizes

Quality of Service

In order to provide acceptable service quality to the clients, the provider $\mathbb{T}$ needs to ensure that the clients are able to download the content at a reasonable rate. While the current design of Floodgate does not have mechanisms in place to ensure such quality of service guarantees, several possible approaches are being actively considered.

The provider $\mathbb{T}$ could start off the content delivery using the traditional 'over the web' process and switch to Floodgate once a threshold level of bandwidth has been used. This would ensure that the initial clients that download the content will not suffer slow speed and that when Floodgate kicks in, there are enough clients out there that already have the full content and would be in a position to share it with the new peers. $\mathbb{T}$ could also proactively insert its own seeders into the swarm, with the intention of ensuring that there are at least some peers that will always be seeding. In order to provide further availability, $\mathbb{T}$ could also consider distributing unrequested content (push model) to (willing) clients in an encrypted format, again with the aim of ensuring that enough seeds exist to kickstart the

seeding process. In these cases, since the content itself is encrypted, there is no danger of the content being available to these bootstrap clients for free. The decryption key would then be sent to the paying client when the actual payment for the content is made to $\mathbb{T}$.

Botnet infiltration

While in theory it is possible for a botnet controller to use compromised machines as peers in the Floodgate network with the aim of collecting large number of micropayment tokens, in practice the closed nature of the network prevents this from happening. Since every peer in the network needs to register with $\mathbb{T}$, registration of a large number of peers by or transfer of a large number of tokens to a single entity would be noticeable and countermeasures can be taken.

ISP & Network neutrality

The recent alleged degrading of P2P traffic by Internet Service Providers [10], especially over the last mile, has brought into question the feasibility of implementing real-world P2P based CDNs. We believe that this is not an issue that can be addressed technically but rather it is a problem involving economical, political and legal intricacies and hence should be handled at those levels.

VI. Related Work

Virtual currency has previously been proposed as an incentive mechanism for P2P systems. Almost all of them, however, are either inefficient or prone to manipulation. Systems like Scrivener [11] for example use a credit system whereby clients can use credits earned (for uploads) to download more content within the same network. The inability to use the credit across networks or exchange it for monetary or other goods-based rewards make such systems less appealing. The micropayment schemes PPay [12] and WhoPay [13] though proposed specifically for P2P-based content delivery network, does not prevent free riding since it allows for short-lived connections to download portion of the content without payment. Floodgate can prevent this by making the client pay first before it is handed over the tokens. That way, even if a dispute arise, $\mathbb{T}$ can act a mediator to resolve it.

Torrent Entertainment Network [2] aims to provide licensed content to paying users by serving Windows DRM protected content. The service provides "reliable download" by seeding their own content and hopes to leverage on the users to provide additional upload bandwidth. However, the users are still bound only by the tit-for-tat mechanism employed by the traditional BitTorrent protocol and is not incentivized in any other way to seed the contents. The use of DRM in itself is orthogonal to the actual protocol used to transfer the content and can be used with Floodgate too.

Dandelion [14] is a content delivery network that uses currency-based incentives in a manner similar to Floodgate. However, in Dandelion after each piece is received the peer needs to contact the server to obtain the decryption key. This could put a heavy load on the server as well as prove to be a bottleneck in the process since the peer has to wait until the decryption key is obtained and integrity-checked before it can download the next piece. Floodgate does not have such a bottleneck. The original proposal also contains a security flaw in which the decryption key is sent back to the peer in plain text. Furthermore, the protocol uses soft time synchronization, which can be exploited by a free-riding attacker, concerns which have not been addressed in the work.

VII. Conclusion

In this paper we have presented the design and implementation of Floodgate, a secure and fair incentive driven protocol for collaborative P2P content delivery networks. It was shown that the protocol is resistant to free-riding and other attacks found in current P2P systems. The performance measurements conducted over the PlanetLab infrastructure showed that our prototype implementation had a low, but non-negligible overhead. As future work, we hope to perform more extensive performance measurements including isolating performance time with respect to transfer and CPU overhead as well as effect of multiple uploads and downloads. We also plan to optimize the implementation as well as investigate the issue of providing quality of service in Floodgate.

References

[1] *Amazon MP3 Downloads*, http://tinyurl.com/ynsdss

[2] *BitTorrent, Inc. Launches the BitTorrent Entertainment Network*, http://tinyurl.com/29j352, Retrieved on 2008-05-17

[3] B. Cohen, *Incentives Build Robustness in BitTorrent*, In proc. of P2P Economics, 2003.

[4] N. Liogkas, R. Nelson, E. Kohler and L. Zhang, *Exploiting BitTorrent For Fun (But Not Profit)*, In proc. of International Workshop on Peer-to-Peer Systems, 2006.

[5] M. Sirivianos, J.H. Park, R.Chen and X. Yang, *Free-riding in BitTorrent Networks with the Large View Exploit*, In proc. of International Workshop on Peer-to-Peer Systems, 2007.

[6] L. Lamport, *Password Authentication with Insecure Communication*, Communications of the ACM, 24(11), pp. 770–772, 1981.

[7] T. Locher, P. Moor, S. Schmid and R. Wattenhofer, *Free Riding in BitTorrent is Cheap*, In proc. of Workshop on Hot Topics in Networks 2006, 2006.

[8] J. Mirkoic, p. Reihe, *A Taxonomy of DDoS Attack and DDoS Defense Mechanisms*, ACM SIGCOMM Computer Communication Review, 34(2), pp. 39–54, 2004.

[9] *PlanetLab: An open platform for developing, deploying, and accessing planetary-scale services*, http://www.planet-lab.org/

[10] *FCC Pressed to Stop Comcast's Internet Blocking*, http://freepress.net/release/297, Retrieved on 2008-05-17

[11] P. Druschel, A. Nandi, T.W.J. Ngan, A. Singh and D. Wallach, *Scrivener: Providing Incentives in Cooperative Content Distribution Systems*, In proc. of Middleware 2005, pp. 270–291, 2005.

[12] B. Yang and H. Garcia-Molina, *PPay: Micropayments for Peer-to-peer Systems*, In proc. of ACM Conference on Computer and Communications Security, pp. 300–310, 2003.

[13] K.Wei, Y.F. Chen, A.J. Smith and B. Vo, *WhoPay: a Scalable and Anonymous Payment system for Peer-to-peer Environments*, In proc. of IEEE International Conference on Distributed Computing Systems, pp. 13, 2006.

[14] D. Sirivianos, J.H. Park , X. Yang and S. Jarecki, *Dandelion: Cooperative Content Distribution with Robust Incentives*, In proc. of USENIX, pp. 157–170, 2007.