

Give-to-Get: Free-riding-resilient Video-on-Demand in P2P Systems

J.J.D. Mol, J.A. Pouwelse, M. Meulpolder, D.H.J. Epema, and H.J. Sips*

Department of Computer Science, Delft University of Technology
P.O. Box 5031, 2600 GA Delft, The Netherlands

ABSTRACT

Centralised solutions for Video-on-Demand (VoD) services, which stream pre-recorded video content to multiple clients who start watching at the moments of their own choosing, are not scalable because of the high bandwidth requirements of the central video servers. Peer-to-peer (P2P) techniques which let the clients distribute the video content among themselves, can be used to alleviate this problem. However, such techniques may introduce the problem of free-riding, with some peers in the P2P network not forwarding the video content to others if there is no incentive to do so. When the P2P network contains too many free-riders, an increasing number of the well-behaving peers may not achieve high enough download speeds to maintain an acceptable service. In this paper we propose Give-to-Get, a P2P VoD algorithm which discourages free-riding by letting peers favour uploading to other peers who have proven to be good uploaders. As a consequence, free-riders are only tolerated as long as there is spare capacity in the system. Our simulations show that even if 20% of the peers are free-riders, Give-to-Get continues to provide good performance to the well-behaving peers. In particular, they show that Give-to-Get performs very well for short videos, which dominate the current VoD traffic on the Internet.

Keywords: multicasting, video streaming, video-on-demand, peer-to-peer, free-riding.

1. INTRODUCTION

Multimedia content such as movies and TV programs can be downloaded and viewed from remote servers using one of three methods. First, with off-line downloading, a pre-recorded file is transferred completely to the user before he starts watching it. Secondly, with live streaming, the user immediately watches the content while it is being broadcast to him. The third method, which holds the middle between off-line downloading and live streaming and which is the one we will focus on, is Video-on-Demand (VoD). With VoD, pre-recorded content is streamed to the user, who can start watching the content at the moment of his own choosing, from the beginning of the video.

VoD systems have proven to be immensely popular. Web sites serving television broadcasts (e.g., BBC Motion Gallery) or user-generated content (e.g., YouTube) draw huge numbers of users. However, to date, virtually all of these systems are centralised, and as a consequence, they require high-end servers and expensive Internet connections to serve their content to their large user communities. Employing decentralized systems such as peer-to-peer (P2P) networks for VoD instead of central servers seems a logical step, as P2P networks have proven to be an efficient way of distributing content over the Internet. The focus of this paper lies in providing a distributed alternative to video sites like YouTube, which serve short, user-generated content. The first contribution of this paper is a P2P VoD algorithm called “Give-to-Get” which uses simple constructs, which is easy to implement and deploy, and which we show to have good performance.

In P2P networks, free-riding is a well-known problem.^{1,2} A free-rider in a P2P network is a peer who consumes more resources than it contributes, and more specifically, in the case of VoD, it is a peer who downloads data but uploads little or no data in return. The burden of uploading is on the altruistic peers, who may be too few in number to provide all peers with an acceptable quality of service. In both live streaming and VoD, peers require a minimal download speed to sustain playback, and so free-riding is especially harmful as the altruistic peers alone may not be able to provide all the peers with sufficient download speeds. Solutions to solve the free-riding problem have been proposed for off-line downloading³ and live streaming.⁴⁻⁶ However, for P2P VoD, no solution yet exists which takes free-riding into consideration. The second contribution of this paper is the design and analysis of the mechanism that makes Give-to-Get free-riding-resilient. In Give-to-Get, peers have to forward (give) the chunks received from a peer to others in order to get more chunks from that peer. By preferring to serve good forwarders, free-riders are excluded in favour of well-behaving peers. When bandwidth in the P2P system becomes scarce, the free-riders will experience a significant drop in the experienced quality of service. Free-riders will thus be able to obtain video data only if there is spare capacity in the system.

*{j.j.d.mol, j.a.pouwelse, m.meulpolder, d.h.j.epema, h.j.sips}@tudelft.nl

Since Give-to-Get is essentially about data distribution, we have designed it to be video-codec agnostic: Give-to-Get can be implemented using any video codec. Give-to-Get splits the video stream into chunks of fixed size, which are to be played at a constant bitrate. Any P2P VoD system also has to make sure that the chunks needed for playback in the immediate future are present. For this purpose, we define a prebuffering policy for downloading the first chunks of a file before playback starts, as well an ordering of the requests for the downloads of the other chunks. We will use the incurred chunk loss rate and the required prebuffering time as the metrics to evaluate the performance of Give-to-Get, which we will report separately for the well-behaving peers and for the free-riders.

The remainder of this paper is organized as follows. In Section 2, we further specify the problem we address, followed by a description of the Give-to-Get algorithm in Section 3. Next, we present our experiments and their results in Section 4. In Section 5, we discuss related work. Finally, we draw conclusions and discuss future work in Section 6.

2. PROBLEM DESCRIPTION

The problem we address in this paper is the design of a P2P VoD algorithm which discourages free-riding. A free-rider is a peer which consumes more resources than it contributes to the P2P system. We will assume that a peer will not try to cheat the system in a different way, such as being a single peer emulating several peers (also called a Sybil attack⁷), or several peers colluding. In this section, we will describe how a P2P VoD system operates in our setting in general. We assume the algorithm to be designed for P2P VoD to be video-codec agnostic, and we will consider the video to be a constant bit-rate stream with unknown boundary positions between the consecutive frames. Similarly to BitTorrent,³ we assume that the video file to be streamed is split up into chunks of equal size, and that every peer interested in watching the stream tries to obtain all chunks. Due to the similarities with BitTorrent, we will use its terminology to describe both our problem and our proposed solution.

A P2P VoD system consists of peers which are downloading the video (leechers) and of peers which have finished downloading and upload for free (seeders). The system starts with at least one seeder. We assume that a peer is able to obtain the addresses of a number of random other peers, and that connections are possible between any pair of peers. To provide all leechers with the video data in a P2P fashion, a multicast tree has to be used for every chunk of the video. Such a multicast tree can be built explicitly or emerge implicitly as the union of paths over which a certain chunk travelled to each peer. While in traditional application-level multicasting, the same multicast tree is created for all chunks and is changed only when peers arrive or depart, we allow the multicast trees of different chunks to be different based on the dynamic behavior of the peers. These multicast trees are not created ahead of time, but rather come into being while chunks are being propagated in the system.

A peer typically behaves in the following manner: It joins the system as a leecher and contacts other peers in order to download chunks of a video. After a prebuffering period, the peer starts playback. When the video has finished playing, the peer will depart. If the peer is done downloading the video before playback is finished, it will stay as a seeder until it departs. We assume that peers can arrive at any time, but that they will start playing the video from the beginning and at a constant speed. Similar to other P2P VoD algorithms like BiToS and popular centralised solutions like YouTube, we do not consider seeking or fast-forwarding. Give-to-Get can be extended to support these operations, but such extensions are outside the scope of this paper. Rewinding can be supported in the player itself without help of Give-to-Get.

3. GIVE-TO-GET

In this section, we will explain Give-to-Get (G2G). First, we will describe how a peer maintains information about other peers in the system in its so-called neighbour list. Then, the way the video pieces are forwarded from peer to peer is discussed. Next, we show in which order video pieces are transferred between peers. Fourth, we will discuss the differences with the related BitTorrent³ and BiToS⁸ protocols. Finally, we will discuss our performance metrics.

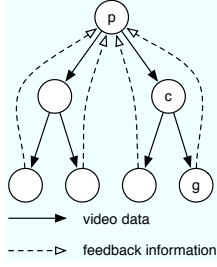
3.1 Neighbour Management

The system we consider consists of peers which are interested in receiving the video stream (leechers) and peers which have obtained the complete video stream and are willing to share it for free (seeders). We assume a peer is able to obtain addresses of other peers uniformly at random. Mechanisms to implement this could be centralised, with a server keeping track of who is in the network, or decentralised, for example, by using epidemic protocols or DHT rings. We view this peer discovery problem as orthogonal to our work, and so beyond the scope of this paper. From the moment a peer joins the system, it will obtain and maintain a list of 10 neighbours in its neighbour list. When a peer is unable to contact 10 neighbours, it will periodically try to discover new neighbours. Once a peer becomes a seeder, it will disconnect from other seeders to avoid maintaining useless connections.

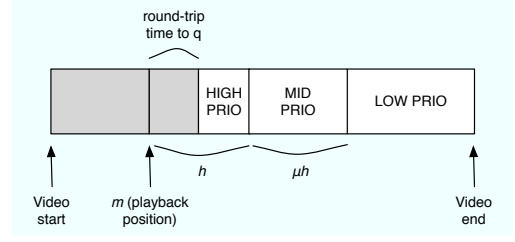
```

choke(all neighbours)
 $N \leftarrow$  all interested neighbours
sort  $N$  on forwarding rank
for  $i = 1$  to  $\min(|N|, 3 + n)$  do
  unchoke( $N[i]$ )
   $b \leftarrow \sum_{k=1}^i$  (our upload speed to  $N[k]$ )
  if  $i \geq 3$  and  $b > \text{UPLINK} * 0.9$  then
    break
  end if
end for
(a) Unchoking algorithm.

```



(b) The feedback connections for an individual peer.



(c) The high-, mid- and low-priority sets in relation to the playback position. The chunks in the grey areas, if requested, will not arrive before their deadline.

Figure 1. The Give-to-Get unchoking, feedback, and piece-picking systems.

3.2 Chunk Distribution

The video data is split up into *chunks* of equal size. As G2G is codec agnostic, the chunk boundaries do not necessarily coincide with frame boundaries. Peers obtain the chunks by requesting them from their neighbours. A peer keeps its neighbours informed about the chunks it has, and decides which of its neighbours is allowed to make requests. A neighbour which is allowed to make requests is *unchoked*. When a chunk is requested by a neighbour, the peer appends it to the send queue for the corresponding connection. Chunks are uploaded using subchunks of 1 Kbyte to avoid delays in the delivery of control messages, which are sent with a higher priority.

Every δ seconds, a peer decides which neighbours are unchoked based on information gathered over the last δ seconds. The neighbours which have shown the best performance will be unchoked, as well as a randomly chosen neighbour (optimistic unchoking). G2G employs a novel unchoking algorithm, described in pseudocode in Figure 1(a). A peer p ranks its neighbours according to decreasing *forwarding ranks*, which is a value representing how well a neighbour is forwarding chunks. The calculation of the forwarding rank is explained below. Peer p unchokes the three highest-ranked neighbours. Since peers are judged by the amount of data they forward, it is beneficial to make efficient use of the available upload bandwidth. To help saturate the uplink, subsequently more neighbours are unchoked until the uplink bandwidth necessary to serve the unchoked peers reaches 90% of p 's uplink. At most n neighbours are unchoked this way to avoid serving too many neighbours at once, which would decrease the performance of the individual connections. The optimal value for n likely depends on the available bandwidth and latency of p 's network connections. In our experiments, we use $n = 2$. To search for better children, p round-robins over the rest of the neighbours and optimistically unchokes a different one of them every 2δ seconds. If the optimistically unchoked neighbour proves to be a good forwarder and ends up at the top, it will be automatically kept unchoked. New connections are inserted uniformly at random in the list of neighbours. The duration of 2δ seconds turns out to be enough for a neighbour to prove its good behaviour. By having a peer upload chunks to only the best forwarders, its neighbours are encouraged to forward the data as much as possible. Peers are not obliged to forward data, but may not be able to receive video data once other peers start to compete for it. This results in a system where free-riders are tolerated only if there is sufficient bandwidth left to serve them.

A peer p ranks its neighbours based on the number of chunks they have forwarded during the last δ seconds. Our ranking procedure consists of two steps. First, the neighbours are sorted according to the decreasing numbers of chunks they have forwarded to other peers, counting only the chunks they originally received from p . If two neighbours have an equal score in the first step, they are sorted in the second step according to the decreasing total number of chunks they have forwarded to other peers. Either step alone does not suffice as a ranking mechanism. If neighbours are ranked solely based on the total number of chunks they upload, good uploaders will be unchoked by all their neighbours, which causes only the best uploaders to receive data and the other peers to starve. On the other hand, if neighbours are ranked solely based on the number of chunks they receive from p and forward to others, peers which are optimistically unchoked by p have a hard time becoming one of the top ranked forwarders. An optimistically unchoked peer q would have to receive chunks from p and hope for q 's neighbours to request exactly those chunks often enough. The probability that q replaces the other top forwarders ranked by p is too low.

Peer p has to know which chunks were forwarded by its children to others. To obtain this information, it cannot ask its children directly, as they could make false claims. Instead, p asks its grandchildren for the behaviour of its children. The children of p keep p updated about the peers they are forwarding to. Peer p contacts these grandchildren, and asks them

which chunks they received from p 's children. This allows p to determine both the forwarding rates of its children as well as the numbers of chunks they forwarded which were originally provided by p . Because peer p ranks its children based on the (amount of) data they forward, the children of p have an incentive to forward as much as possible to obtain a high rank. Figure 1(b) shows an example of the flow of feedback information. Peer p has unchoked two other peers, amongst which peer c . Peer c has peer g unchoked. Information about the amount of video data uploaded by c to g is communicated over the dashed arrow back to p . Peer p can subsequently rank child c based on this information. Note that a node c has no incentive to lie about the identities of its children, because only its actual children will provide feedback about c 's behaviour.

3.3 Chunk Picking

A peer obtains chunks by issuing a request for each chunk to other peers. A peer thus has to decide in which order it will request the chunks it wants to download; this is called *chunk picking*. When a peer p is allowed by one of its neighbours to make requests to it, it will always do so if the neighbour has a chunk p wants. We associate with every chunk and every peer a *deadline*, which is the latest point in time the chunk has to be present at the peer for playback. As long as p has not yet started playback, the deadline of every chunk at p is infinite. Peer p wants chunk i from a neighbour q if the following conditions are met: a) q has chunk i , b) p does not have chunk i and has not previously requested it, and c) it is likely that chunk i arrives at p before its deadline. Peer p will never request pieces it does not want. Because peers keep their neighbours informed about the chunks they have, the first two rules are easy to check. To estimate whether a chunk will arrive on time, p keeps track of the response time of requests. This response time is influenced by the link delay between p and q as well as the amount of traffic from p and q to other peers. Peers can submit multiple requests in parallel in order to fully utilise the links with their neighbours.

When deciding the order in which chunks are picked, two things have to be kept in mind. First, it is necessary to provide the chunks in-order to the video player. Secondly, to achieve a good upload rate, it is necessary to obtain enough chunks which are wanted by other peers. The former favours downloading chunks in-order, the latter favours downloading rare chunks first. To balance between these two requirements, G2G employs a hybrid solution by prioritizing the chunks that have yet to be played back. Let m be the playback position of peer p , or 0 if p has not yet started playback. Peer p will request chunk i on the first match in the following list of sets of chunks (see Figure 1(c)):

- *High priority*: $m \leq i < m + h$. If p has already started playback, it will pick the lowest such i , otherwise, it will pick i rarest first.
- *Mid priority*: $m + h \leq i < m + (\mu + 1)h$. Peer p will choose such an i rarest first.
- *Low priority*: $m + (\mu + 1)h \leq i$. Peer p will choose such an i rarest first.

In these definitions, h and μ are parameters which dictate the amount of clustering of chunk requests in the part of the video yet to be played back. A peer picks rarest-first based on the availability of chunks at its neighbours. Among chunks of equal rarity, i is chosen uniformly at random. During playback, the chunks with a tight deadline are downloaded first and in-order (the high priority set). The mid-priority set makes it easier for the peer to complete the high-priority set in the future, as the (beginning of the) current mid-priority set will be the high-priority set later on. This lowers the probability of having to do in-order downloading later on. The low-priority set will download the rest of the chunks using rarest-first both to collect chunks which will be forwarded often because they are wanted by many and to increase the availability of the rarest chunks. Also, the low priority set allows a peer to collect as much of the video as fast as possible.

3.4 Differences between Give-to-Get, BitTorrent and BiToS

In our experiments, we will compare the performance of G2G to that of BiToS.⁸ BiToS is a P2P VoD algorithm which, like G2G, is inspired by BitTorrent. For BiToS, we will use the optimal settings as derived in the paper where BiToS is introduced.⁸ The major differences between G2G, BiToS and BitTorrent lie in the chunk-picking policy, the choking policy and the prebuffering policy. In BiToS, two priority sets are used: the high-priority set and the remaining-pieces set. The high-priority set is defined to be 8% of the video length. Peers request pieces from the high-priority set 80% of the time, and from the remaining-pieces set 20% of the time. The rarest-first policy is used in both cases, with a bias towards earlier chunks if they are equally rare. In contrast, G2G uses three priority sets. In-order downloading is used in the high-priority set once playback has started, and in the mid- and low-priority sets, the rarest-first policy chooses at random between pieces of equal rarity. BitTorrent is not designed for VoD and thus does not define priority sets based on the playback position.

The choking policy determines which neighbours are allowed to request pieces, which defines the flow of chunks through the P2P network. BiToS, like BitTorrent, is based on tit-for-tat, while G2G is not. In tit-for-tat, a peer a will allow a peer

b to make requests for chunks if b proved to be one of the top uploaders to a . In contrast, a peer a in G2G will allow b to make requests if b proves to be one of the top forwarders to others (this set of others can include a). Tit-for-tat works well in an off-line download setting (such as BitTorrent) where peers have enough chunks they can exchange. However, it is less suitable for VoD because peers in VoD bias their interests on downloading the chunks close after their playback position, and are not interested in chunks before their playback position. Two peers subsequently either have overlapping interests if their playback positions are close, or the peer with the earliest playback position is interested in the other's chunks but not vice-versa. One-sided interests are the bane of tit-for-tat systems. In BiToS, peers download pieces outside their high-priority set 20% of the time, relaxing the one-sided interests problem somewhat, as all peers have a mutual interest in obtaining the end of the video this way. The choking policy in G2G consists of unchoking neighbours which have proven to be good forwarders. Subsequently, the requirement to exchange data between peers, and thus to have interests in each other's chunks, is not present in G2G.

3.5 Performance Metrics

For a peer to view a video clip in a VoD fashion, two conditions must be met to provide a good quality of service. First, the start-up delay must be small, and secondly, the chunk loss must be low to provide good playback quality. If either of these conditions is not met, it is likely the user was better off downloading the whole clip before viewing it. In G2G, we say a chunk is lost if a peer cannot request it in time from one of its neighbours, or if the chunk was requested but did not arrive in time. The concept of buffering the beginning of the video clip before starting playback is common to most streaming video players. We will assume that once prebuffering is finished and playback is started, the video will not be paused or slowed down. In general, the amount of prebuffering is a trade-off between having a short waiting period and having low chunk loss during playback.

We define the prebuffering time as follows. First, a peer waits until it has the first h chunks (the initial high-priority set) available to prevent immediate chunk loss. Then, it waits until the expected remaining download time is less than the duration of the video. The expected remaining download time is extrapolated from the download speed so far, with a 20% safety margin. This margin allows for short or small drops in download rate later on, and will also create an increasing buffer when the download rate does not drop. Drops in download rate can occur when a parent of a peer p adopts more children or replaces p with a different child due to p 's rank or to optimistic unchoking. In the former case, the uplink of p 's parent has to be shared by more peers, and in the latter case, p stops receiving anything from that particular parent. When and how often this will occur depends on the behaviour of p and its neighbours, and is hard to predict. The safety margin in the prebuffering was added to protect against such behaviour. It should be noted that other VoD algorithms which use BitTorrent-like unchoking mechanisms (such as BiToS⁸) are likely to suffer from the same problem. In order to keep the prebuffering time reasonable, the average upload speed of a peer in the system should thus be at least the video bitrate plus a 20% margin. If there is less upload capacity in the system, peers both get a hard time obtaining all chunks and are forced to prebuffer longer to ensure the download will be finished before the playback is.

Once a peer has started playback, it requires chunks at a constant rate. The chunk loss is the fraction of chunks that have not arrived before their deadlines, averaged over all the playing peers. When reporting the chunk loss, a 5-second sliding window average will be used to improve the readability of the figures. Neither BiToS nor BitTorrent define a prebuffering policy, so to be able to make a fair comparison between BiToS and G2G in our experiments, we will use the same prebuffering policy for BiToS as for G2G. BiToS uses a larger high-priority set (8% of the video length, or 24 seconds for the 5-minute video we will use in our experiments), making it unfair to let peers wait until their full high-priority set is downloaded before playback is started. Instead, like in G2G, we wait until the first $h = 10$ seconds are downloaded.

4. EXPERIMENTS

In this section we will present our experimental setup as well as the results of two experiments. In the first experiment, we measure the default behaviour with well-behaving peers and in the second experiment we let part of the system consist of free-riders. In both cases, we will compare the performance of G2G and BiToS.

4.1 Experimental Setup

Our experiments were performed using a discrete-event simulator, emulating a network of 500 peers which are all interested in receiving the video stream. The network is assumed to have no bottlenecks except at the peers. Packets are sent using TCP with a 1500 byte MTU and their delivery is delayed in case of congestion in the uplink of the sender or the downlink of the receiver. Each simulation starts with one initial seeder, and the rest of the peers arrive according to a Poisson

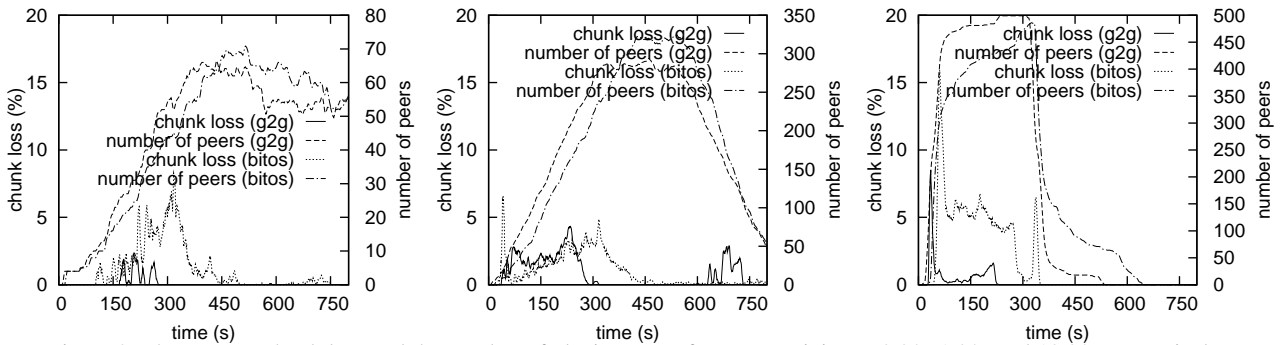


Figure 2. The average chunk loss and the number of playing peers for peers arriving at 0.2/s, 1.0/s, and 10.0/s, respectively.

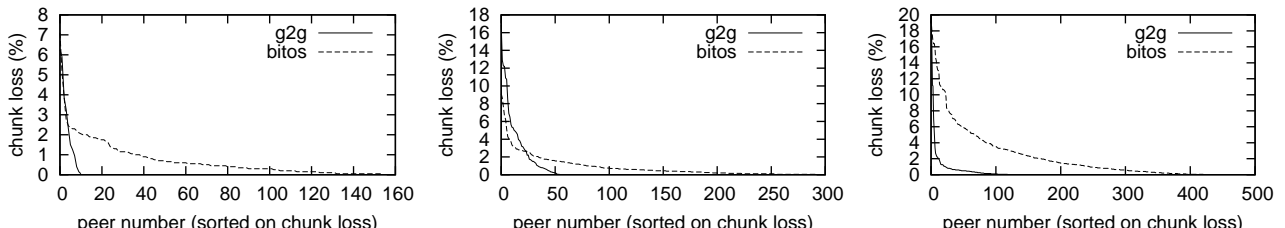


Figure 3. The distribution of the chunk loss over the peers for peers arriving at 0.2/s, 1.0/s, and 10.0/s, respectively.

process. Unless stated otherwise, peers arrive at a rate of 1.0/s, and depart when playback is finished. When a peer is done downloading the video stream, it will therefore become a seeder until the playback is finished and the peer departs.

We will simulate a group of peers with asymmetrical bandwidth capacities, which is typical for end-users on the Internet. Every peer has an uplink capacity chosen uniformly at random between 0.5 and 1.0 Mbit/s. The downlink capacity of a peer is always four times its uplink capacity. The round-trip times between peers vary between 100 ms and 300 ms[†]. A peer reconsiders the behaviour of its neighbours every $\delta = 10$ seconds, which is a balance between keeping the overhead low and allowing neighbour behaviour changes (including free-riders) to be detected. The high-priority set size h is defined to be the equivalent of 10 seconds of video. The mid-priority set size is $\mu = 4$ times the high-priority set size.

A 5-minute video of 0.5 Mbit/s is cut up into 16 Kbyte chunks (i.e., 4 chunks per second on average) and is being distributed from the initial seeder with a 2 Mbit/s uplink. We will use the prebuffering time and the chunk loss as the metrics to assess the performance of G2G. The actual frame loss depends on the video codec and the dependency between encoded frames. Because G2G is codec-agnostic, it does not know the frame boundaries in the video stream and thus, we cannot use frame loss as a metric. In all experiments, we will compare the performance of G2G and a BiToS system. We will present the results of a representative sample run to show typical behaviour. We will use the same arrival patterns, neighbour sets and peer capabilities when comparing the performance of G2G and BiToS.

4.2 Default Behaviour

In the first experiment, peers depart only when their playback is finished, and there are no free-riders. We do three runs, letting peers arrive at an average rate of 0.2/s, 1.0/s, and 10.0/s, respectively, to that of BiToS when using the same arrival pattern and network configuration. In Figure 2, the number of playing peers in the system is shown as well as the average percentage of chunk loss. In the left and middle graphs, peers start departing before all of them have arrived, resulting in a more or less stable number of peers for a certain period that ends when all peers have arrived. In the right graph, all peers have arrived within approximately 50 seconds, after which the number of peers is stable until all of them are done playing. As long as the initial seed is the only seed in the system, peers experience some chunk loss. Because all peers are downloading the video, there is much competition for bandwidth. Once some peers are done downloading the video, they can seed it to others and after a short period of time, no peer experiences any chunk loss at all. In effect, the seeders form a content distribution network aided by the peers which continue to forward chunks to each other.

Figure 3 shows the distribution of the chunk loss for each arrival rate across the peers, sorted decreasingly. At all three rates, the chunk loss is concentrated on a small number of peers, but much more so for G2G than for BiToS. The graphs for

[†]These figures are realistic for broadband usage within the Netherlands. The results are nevertheless representative, because the overhead of G2G is low compared to the video bandwidth.

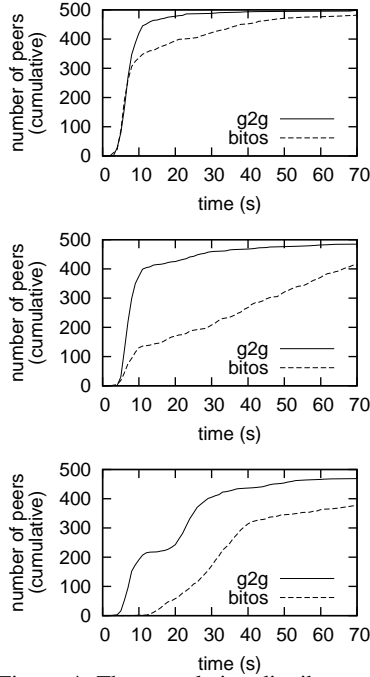


Figure 4. The cumulative distribution of the prebuffering time for peers arriving at 0.2/s, 1.0/s, and 10.0/s, respectively.

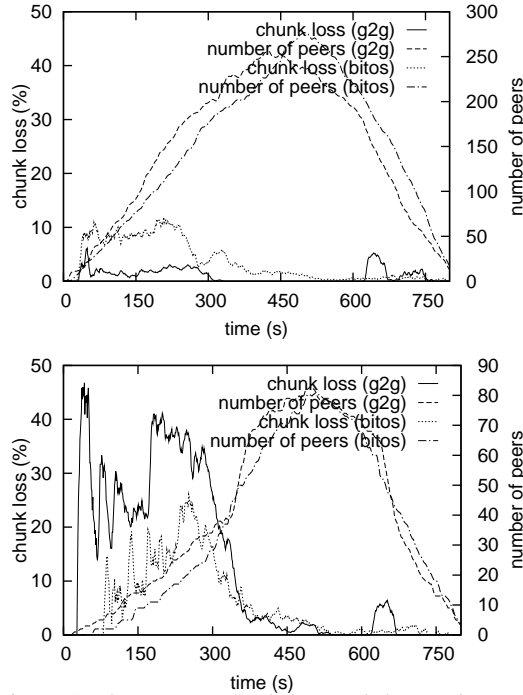


Figure 5. The average chunk loss and the number of playing peers, for well-behaving peers (top) and free-riders (bottom).

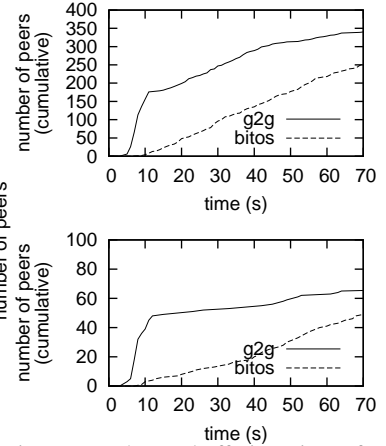


Figure 6. The prebuffering time, for well-behaving peers (top) and free-riders (bottom).

G2G are mostly below those of BiToS, implying that when considering chunk loss, most peers are better off using G2G. A more sophisticated playback policy, such as allowing the video stream to pause for rebuffering, could potentially alleviate the heavy losses that occur for some peers using either algorithm.

Figure 4 shows the cumulative distribution of the required prebuffering time, which increases with the arrival rate. At higher arrival rates, it takes an increasing amount of time before the initial pieces are spread across the P2P network. A peer has to wait longer before its neighbours have obtained any pieces, and thus the average prebuffering time increases. The required prebuffering time is longer in BiToS, which can be explained by the fact that the high-priority set is large (24 seconds) and is downloaded with the rarest-first policy, so it takes longer to obtain the initial 10 seconds of the video.

4.3 Free-riders

In the second experiment, we add free-riders to the system by having 20% of the peers not upload anything to others. Figure 5 shows the average chunk loss separately for the well-behaving peers and the free-riders. The well-behaving peers are affected by the presence of the free-riders, and experience a higher chunk loss than in the previous experiment. A slight performance degradation is to be expected, as well-behaving peers occasionally upload to free-riders as part of the optimistic unchoking process. Without any means to detect free-riders before serving them data, losing a bit of performance due to the presence of free-riders is unavoidable. When using G2G, the well-behaving peers lose significantly fewer chunks when compared to BiToS, and free-riders suffer higher amounts of chunk loss.

The required prebuffering times for both groups are shown in Figure 6. Both groups require more prebuffering time than in the previous experiment: the well-behaving peers require 33 seconds when using G2G, compared to 14 seconds when free-riders are not present. Because the free-riders have to wait for bandwidth to become available, they either start early and suffer a high chunk loss, or they have a long prebuffering time. In the shown run, the free-riders required 89 seconds of prebuffering on average when using G2G.

5. RELATED WORK

In P2Cast,⁹ P2VoD¹⁰ and OBN,¹¹ peers are grouped according to similar arrival times. The groups forward the stream within the group or between groups, and turn to the source if no eligible supplier can be found. In all three algorithms, a peer can decide how many children it will adopt, making the algorithms vulnerable to free-riding.

Our G2G algorithm borrows the idea of bartering for chunks of a file from BitTorrent,³ which is a very popular P2P protocol for off-line downloading. In BitTorrent, the content is divided into equally sized chunks which are exchanged by peers on a tit-for-tat basis. In deployed BitTorrent networks, the observed amount of free-riding is low¹² (however, Locher et al.² have shown that with a specially designed client, free-riding in BitTorrent is possible without a significant performance impact for the free-riding peer). Of course, the performance of the network as a whole suffers if too many peers free-ride. The BitTorrent protocol has been adapted for VoD by both BASS¹³ and BiToS.⁸ In BASS,¹³ all peers connect to a streaming server, and use the P2P network to help each other in order to shift the burden off the source. BASS is thus a hybrid between a centralised and a distributed VoD solution. The differences between G2G and BiToS are described in Section 3.4. Annapureddy et al.¹⁴ propose a comprehensive VoD protocol in which the video stream is divided into segments, which are further divided into blocks (chunks). Peers download segments sequentially from each other and spend a small amount of additional bandwidth prefetching the next segment. Topology management is introduced to connect peers which are downloading the same segment, and network coding is used within each segment to improve resilience. However, their algorithm assumes peers are honest and thus free-riding is not considered. Using indirect information as in G2G to judge other peers is not new. EigenTrust¹⁵ defines a central authority to keep track of all trust relationships in the network. Lian et al.¹⁶ generalise the use of indirect information to combat collusion in tit-for-tat systems.

As is common in streaming video players, G2G buffers the beginning of the video clip (the prefix) before playback is started. Sen et al.¹⁷ propose a scheme to let proxy servers do this prefix caching for a centralised VoD service, thus smoothing the path from server to client.

6. CONCLUSIONS AND FUTURE WORK

We have presented Give-to-Get (G2G) a P2P VoD algorithm which discourages free-riding and introduces a novel chunk-picking policy. We evaluated the performance of Give-to-Get by conducting tests under various conditions as well as comparing its performance to that of BiToS.⁸ If free-riders are present, they suffer heavy performance penalties if upload bandwidth in the system is scarce. Once enough peers have downloaded the full video, there is enough upload bandwidth to sustain free-riders without compromising the well-behaving peers. We plan to enhance G2G by implementing seek and fast-forward functionality, by performing a sensitivity analyses on the parameters of G2G, and by adding a trust system to enhance the quality of the feedback from peers in G2G, and to detect collusion attacks. In addition, we plan to incorporate G2G into Tribler,¹⁸ our social-based P2P system, to allow real-world measurements of G2G's performance.

REFERENCES

1. E. Adar and B. Huberman, "Freeriding on Gnutella," *First Monday* 5(10), 2000.
2. T. Locher, P. Moor, S. Schmid, and R. Wattenhofer, "Free Riding in BitTorrent is Cheap," in *HotNets-V*, 2006.
3. B. Cohen, "BitTorrent," <http://www.bittorrent.com/>.
4. I. Keidar, R. Melamed, and A. Orda, "EquiCast: Scalable Multicast with Selfish Users," in *PODC 2006*, pp. 63–71.
5. H. Li, A. Clement, E. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin, "BAR Gossip," in *OSDI'06*, pp. 191–206.
6. J. Mol, D. Epema, and H. Sips, "The Orchard Algorithm: P2P Multicasting without Free Riding," in *P2P2006*, pp. 275–282.
7. J. Douceur, "The Sybil Attack," in *IPTPS'02*.
8. A. Vlavianos, M. Iliofotou, and M. Faloutsos, "BiToS: Enhancing BitTorrent for Supporting Streaming Applications," in *IEEE Global Internet Symposium*, 2006.
9. Y. Guo, K. Suh, J. Kurose, and D. Towsley, "P2Cast: P2P Patching Scheme for VoD Services," in *WWW2003*, pp. 301–309.
10. T. Do, K. Hua, and M. Tantaoui, "P2VoD: Providing Fault Tolerant Video-on-Demand Streaming in Peer-to-Peer Environment," in *Proc. of the IEEE Intl. Conf. on Communications*, 3, pp. 1467–1472, 2004.
11. C. Liao, W. Sun, C. King, and H. Hsiao, "OBN: Peering Finding Suppliers in P2P On-demand Streaming Systems," in *ICPADS*, 1, pp. 8–15, 2006.
12. N. Andrade, M. Mowbray, A. Lima, G. Wagner, and M. Ripeanu, "Influences on Cooperation in BitTorrent Communities," in *Proc. ACM SIGCOMM*, pp. 111–115, 2005.
13. C. Dana, D. Li, D. Harrison, and C.-N. Chuah, "BASS: BitTorrent Assisted Streaming System for Video-on-Demand," in *MMSP 2005*, pp. 1–4.
14. S. Annapureddy, S. Guha, and C. Gkantsidis, "Is High-Quality VoD Feasible using P2P Swarming?," in *WWW2007*, pp. 903–911.
15. P. Ganesan and M. Seshadri, "The EigenTrust Algorithm for Reputation Management in P2P Networks," in *WWW2003*, pp. 446–457.
16. Q. Lian, Y. Peng, M. Yang, Z. Zhang, Y. Dai, and X. Li, "Robust Incentives via Multi-level Tit-for-tat," in *IPTPS'06*.
17. S. Sen, J. Rexford, and D. Towsley, "Proxy Prefix Caching for Multimedia Streams," in *Proc. of IEEE INFOCOM*, 3, pp. 1310–1319, 1999.
18. J. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. Epema, M. Reinders, M. van Steen, and H. Sips, "Tribler: A Social-Based Peer-to-Peer System," in *Concurrency and Computation: Practice and Experience (to appear)*,