

# Give-to-Get: An Algorithm for P2P Video-on-Demand

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

E-mail: {j.j.d.mol,j.a.pouwelse,m.meulpolder,d.h.j.epema,h.j.sips}@tudelft.nl

## 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. Also, they show that Give-to-Get performs best if the video that is distributed is short.*

## 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.

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.

In P2P networks, free-riding is a well-known problem [1, 9]. 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 downloading [3] and live streaming [6, 7]. However, for P2P VoD, no solution yet exists which takes free-riding into consideration. Therefore, in this paper we propose an algorithm called “Give-to-Get” for VoD in P2P systems, which assumes videos to be split up into chunks of a fixed size. In Give-to-Get, peers have to upload (give) the chunks received from other peers in order to get additional chunks from those peers. By preferring to serve good forwarders, free-riders are excluded in favour of well-behaving peers. Free-riders will thus be able to obtain video data only if there is spare capacity in the system.

Apart from the problems introduced by free-riders, a P2P VoD system also has to take care that the chunks needed for playback in the immediate future are present. For this pur-

pose, we define a chunk prebuffering policy for downloading the first chunks of a file before playback starts, as well as an ordering of the requests for the downloads of the other chunks. We will use the required prebuffering time and the incurred chunk loss rate as the metrics to evaluate the performance of Give-to-Get, which we will report separately for the free-riders and for the well-behaving peers.

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

## 2 Related Work

An early algorithm for distributed VoD is Chaining [10], in which the video data is distributed over a chain of peers. Such a solution works well for a controlled environment, but is less suitable for P2P networks. In P2Cast [5], P2VoD [4] and OBN [8], 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 Give-to-Get algorithm borrows the idea of bartering for chunks of a file from BitTorrent [3], which is a very popular P2P system 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 low [2]. However, a recent study [9] has 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. In BiToS [11], the BitTorrent protocol is adapted for VoD. Peers focus most of their tit-for-tat chunk trading on their high-priority set, which consists of the chunks which are due for playback soon. Because the high-priority sets of different peers do not necessarily overlap due to differences in their arrival times, tit-for-tat is not always possible. Also, in [11], the prebuffering time required by peers is not studied, making it hard to judge the obtained quality of service.

BAR Gossip [7] and EquiCast [6] are both solutions for live streaming which combat free-riding, but they are not aimed at VoD and they require trusted nodes to police the network and a central server to monitor and maintain the network, respectively.

## 3 Problem Description

The problem we address in this paper is the design of a P2P VoD algorithm which discourages free-riding. In this section, we will describe how a P2P VoD system operates in our setting in general.

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. 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. An important requirement is that we want our solution to be resilient to free-riding.

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, and after the peer has finished downloading, it will remain a seeder until it departs. A peer departs when the video has finished playing, or earlier if the user aborts playback.

We assume the algorithm to be designed for P2P VoD to be video-codec agnostic, and so we will consider the video to be a constant bit-rate stream with unknown boundary positions between the consecutive frames.

## 4 Give-to-Get

In this section, we will explain *Give-to-Get*, our P2P VoD algorithm which discourages free-riding. 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. Finally, we will discuss when a peer decides it can start playing the video.

## 4.1 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 no 10 neighbours can be contacted, the peer periodically tries to discover new neighbours. To maintain a useful set of neighbours, a peer disconnects from any neighbour if no video data has been sent in either direction over the last 30 seconds. Also, once a peer becomes a seeder, it disconnects from other seeders.

## 4.2 Chunk Distribution

In this section we will present the way peers forward chunks of a video file to each other. We first discuss the general mechanism, and subsequently explain the way peers rank their neighbours in this mechanism.

### 4.2.1 Forwarding Chunks

The video data is split up into *chunks* of equal size. As Give-to-Get 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, the peer appends it to the send queue for that connection. Chunks are uploaded using sub-chunks of 1 Kbyte to avoid delays in the delivery of control messages, which are sent with a higher priority.

Every  $\delta$  seconds, a peer decides which neighbours are unchoked based on information gathered over the last  $\delta$  seconds. Algorithm 1 shows the pseudocode of how these decisions are made. A peer  $p$  ranks its neighbours according to their *forwarding ranks*, which represent how well a neighbour is forwarding chunks. The calculation of the forwarding rank is explained in the next section. Peer  $p$  unchokes the best three forwarders. 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

---

### Algorithm 1 Unchoking algorithm (optimistic unchoking not included)

---

```

choke(all neighbours)
 $N \leftarrow$  all interested neighbours
sort  $N$  on forwarding rank
for  $i = 1$  to  $\min(|N|, 3)$  do
    unchoke( $N[i]$ )
end for
for  $i = 4$  to  $\min(|N|, 4 + \epsilon)$  do
     $b \leftarrow \sum_{k=1}^{i-1}$  (upload speed to  $N[k]$ )
    if  $b > \text{UPLINK} * 0.9$  then
        break
    end if
    unchoke( $N[i]$ )
end for

```

---

the uplink, more neighbours are unchoked until the uplink bandwidth necessary to serve the unchoked peers reaches 90% of  $p$ 's uplink. At most  $\epsilon$  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  $\epsilon$  likely depends on the available bandwidth and latency of  $p$ 's network connection. In our experiments, we use  $\epsilon = 2$ .

To search for better children,  $p$  round-robins over the rest of the neighbours and optimistically unchokes a different one of them every  $2\delta$  seconds. If the optimistically unchoked peer 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 optimistic unchoke list. The duration of  $2\delta$  seconds is enough for a neighbour to prove its good behaviour.

By only uploading chunks to the best forwarders, its neighbours are encouraged to forward the data as much as possible. While peers are not obliged to forward their data, they run the danger of not being 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.

### 4.2.2 Forwarding Rank

A peer  $p$  ranks its neighbours based on the number of chunks they have forwarded during the last  $\delta$  seconds. Our ranking procedure consists of two steps:

1. 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  $p$ .
2. If two neighbours have an equal score in the first step, they are sorted 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 neighbours to others. To obtain this information, it cannot ask its neighbours directly, as they could make false claims. Instead,  $p$  asks its grandchildren for the behaviour of its children. The neighbours 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 neighbours. This allows  $p$  to determine both the forwarding rates of its neighbours as well as the numbers of chunks they forwarded which were originally provided by  $p$ .

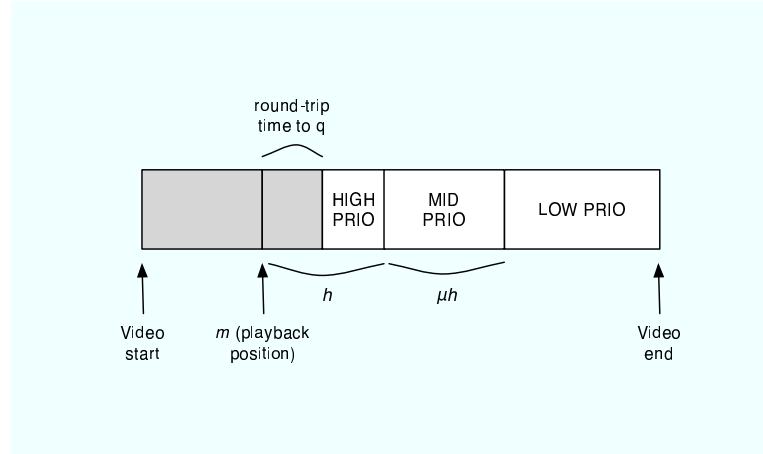
### 4.3 Chunk Picking

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:

- $q$  has chunk  $i$ ,
- $p$  does not have chunk  $i$  and has not previously requested it,
- It is likely that chunk  $i$  arrives at  $p$  before its deadline.

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



**Figure 1. The high-, mid- and low-priority sets in relation to the playback position and the average round-trip time to  $p$ 's neighbours.**

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, Give-to-Get employs a hybrid solution by prioritizing the chunks that have yet to be played back. Let  $m$  be the minimum chunk number peer  $p$  wants from a neighbour  $q$ . Peer  $p$  will request chunk  $i$  on the first match in the following list of sets of chunks (see Figure 1):

- *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  $\mu$  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.

This ordering ensures in-order downloading in the high-priority set (when chunk deadlines are due). The mid-priority set downloads the future high-priority set in advance using rarest-first. 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.

## 4.4 Video Playback

The primary objective of a peer is to view the offered video clip. 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 to provide the user with a streaming experience. Second, the chunk loss must be low to provide good playback quality. If either of these criteria is not met, it is likely the user was better off downloading the whole clip before viewing it.

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. This makes the prebuffering period an important metric in VoD.

In Give-to-Get, we decide on the prebuffering time as follows. First, the peer waits until it has the first  $h$  chunks (the first high-priority set) available to prevent immediate chunk loss. Then, it waits until the remaining download time is less than the duration of the video plus 20% overhead. This overhead 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.

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 20% overhead. 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.

## 5 Experiments

In this section we will present our experimental setup as well as the experiments for assessing the performance of the Give-to-Get algorithm. We will present the results of four experiments. In the first experiment, we measure the default behaviour with well-behaving peers. In the second experiment, we measure the performance when a part of the system consists of free-riders. Next, we let peers depart before they have finished watching the video. In the last experiment, we increase the movie length.

### 5.1 Experimental Setup

Our experiments were performed using a discrete-event simulator, emulating a network of 500 peers. Each simulation starts with one initial seeder, and the rest of the peers arrive according to a Poisson 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. Every peer has an uplink

capacity chosen uniformly at random between 0.5 and 1.0 Mbit/s. The uplink capacity of a peer is evenly shared between the connections and is considered to be the bottleneck between any pair of peers. The round-trip times between pairs of 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 required prebuffering time and the chunk loss as the metrics to assess the performance of Give-to-Get. We will present separate results for the free-riders and the well-behaving peers.

### 5.2 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. Figures 2, 3 and 4 show the results of a typical run.

In Figure 2(a), 3(a) and 4(a), the percentage of chunk loss is shown over time, as well as the number of playing peers and the number of seeders in the system. As long as the initial seed is the only seed in the system, peers experience some chunk loss. Once 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. A comparison the graphs of the three arrival rates reveals that performance is better at high and low arrival rates compared to the performance at 1.0/s. At low arrival rates, the maximum number of peers that can be playing at the same time is low. In a small system, the initial seed has a relatively high impact on the performance. When the arrival rate is high, peers have neighbours which are interested in the same chunks and thus overlapping mid priority sets. As a result, the number of neighbours which have interesting pieces increases for all peers. In contrast, if two peers do not have an overlapping mid priority set, they have a highly biased interest in each others' pieces. The peer with the lower playback position is likely to be interested in pieces of the peer with the higher playback position, but not the other way around. Further experiments have shown that for the parameters we use in this paper, the arrival rate of 1.0/s produces the worst performance.

Figures 2(b), 3(b) and 4(b), show the cumulative distribution of the required prebuffering time. Recall that peers start playback as soon as they have obtained the first 10

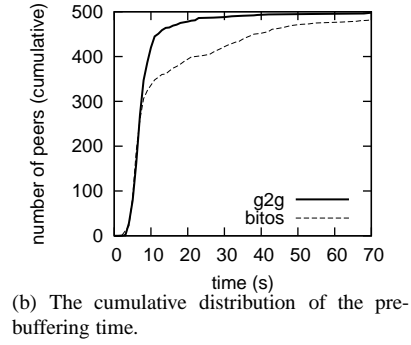
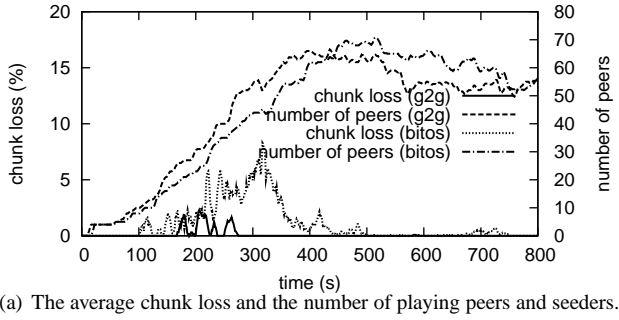


Figure 2. A sample run with peers arriving at a rate of 0.2 per second.

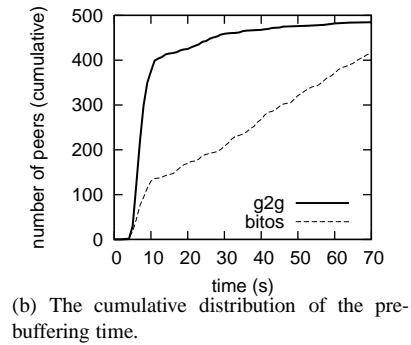
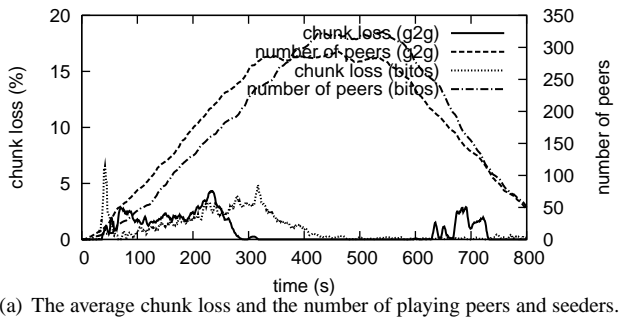


Figure 3. A sample run with peers arriving at a rate of 1.0 per second.

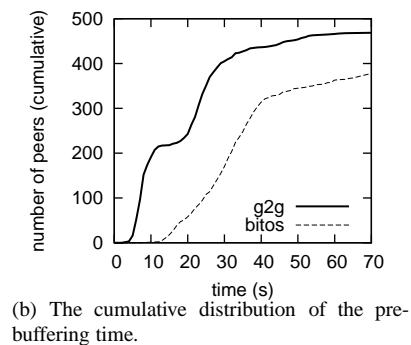
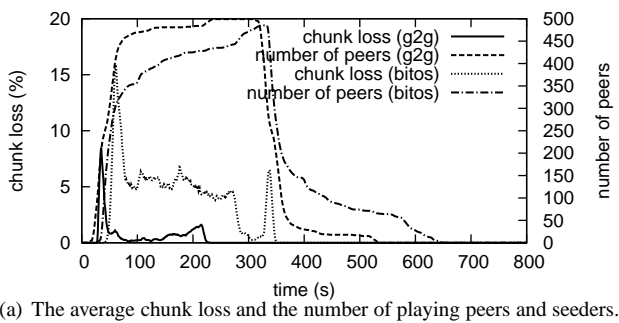


Figure 4. A sample run with peers arriving at a rate of 10.0 per second.

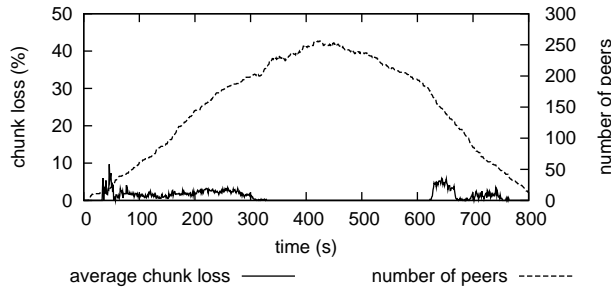
seconds of video, and the download speed is 1.2 times the video playback speed. The measured prebuffering times reflect this, as all peers require at least 10 seconds before they can start playback. The arrival rate influences the prebuffering time as can be seen by comparing the individual graphs. At high arrival rates, there are many peers in the system before the first few peers have obtained the initial 10 seconds of video. Due to the high competition, it takes longer for all peers to obtain these pieces, obtain enough download speed and start playback.

The rest of the experiments assume an arrival rate of

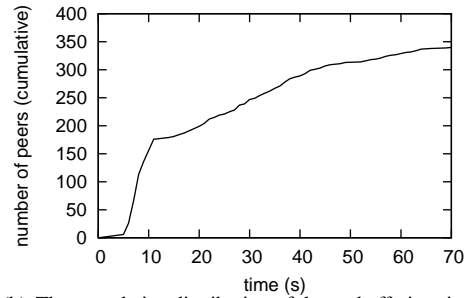
1.0/s. In that case, Figure 3(b) shows that half of them can start playback within 14 seconds, and 90% of them can start playback within 41 seconds. We present these numbers to allow a comparison with the other experiments.

### 5.3 Free-riders

In the second experiment, we add free-riders to the system by having 20% of the peers not upload anything to others. The results of this experiment are shown for the well-behaving peers in Figures 5(a) and 5(b). The well-behaving

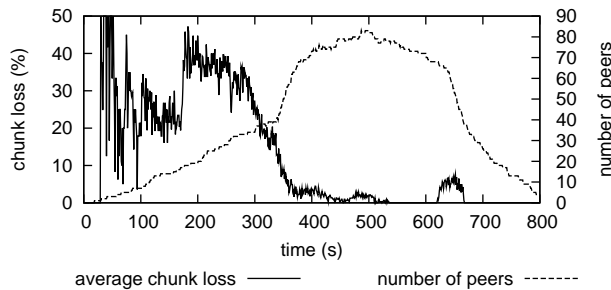


(a) The average chunk loss and the number of playing peers and seeders.

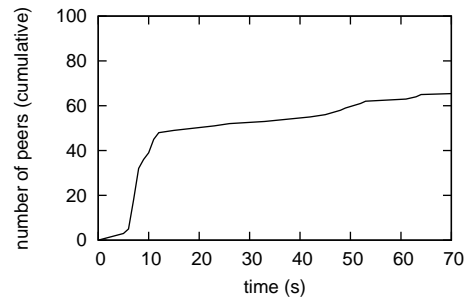


(b) The cumulative distribution of the prebuffering time.

**Figure 5. The performance of the well-behaving peers if 20% of the peers free-ride.**



(a) The average chunk loss and the number of playing peers and seeders.



(b) The cumulative distribution of the prebuffering time.

**Figure 6. The performance of the free-riders if 20% of the peers free-ride.**

peers are affected by the presence of the free-riders, mostly by experiencing a bit more chunk loss and but also by requiring a bit more prebuffering time. A slight performance degradation is to be expected, as well-behaving peers occasionally upload to free-riders as part of the optimistic unchoking process. Any chunk sent to a free-rider is not sent to another well-behaving peer instead. Half of the peers require at most 30 seconds of prebuffering, and 90% of them require at most 53 seconds, which is substantially more than in a system without free-riders, as shown in Figure 3(b).

For the free-riders, the performance is much worse as can be seen in Figures 6(a) and 6(b). When peers are still arriving and no seeders are present, the leechers have to compete and the free-riders suffer from a high chunk loss. Once seeders start to become available, bandwidth becomes more abundant, and well-behaving peers are sufficiently served, the free-riders as well as the well behaving peers suffer little to no chunk loss. 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, half of the free-riders required up to 52 seconds of prebuffering time, but 90% required at most 259 seconds.

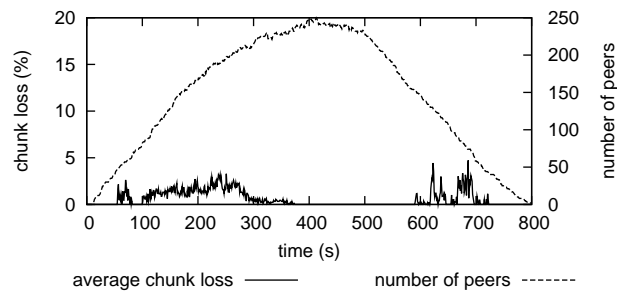
#### 5.4 Early Departures

In the third experiment, we assess the impact of peers departing before they finish playback. Peers that depart early have a lower probability of having downloaded, and thus forwarded, the higher-numbered chunks. Also, they have a lower probability of seeding. Thus, the availability of the highest-numbered chunks and seeders in general decreases.

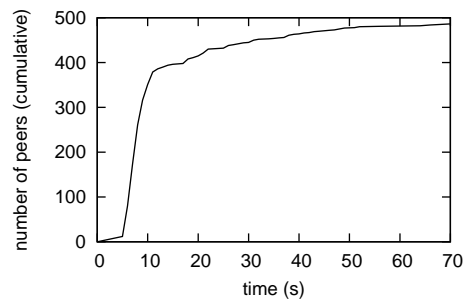
We let half of these peers depart during playback, at a point chosen uniformly at random. In Figure 7(a), we show the chunk loss results, and in Figure 7(b), we show the prebuffering times required by the peers. The chunk loss is certainly higher than in the previous tests, but is still within an acceptable range. Half of the peers start playback within 19 seconds, and 90% of them start playback within 41 seconds, which is slightly worse than when peers do not depart before playback is finished.

#### 5.5 Increased Video Length

In the last experiment, we measure the effect of increasing the video length, as well as the effect of increasing the  $\mu$  variable, which controls the size of the mid priority set. The results for videos with a length of 5, 10 and 30 minutes are shown in Figure 8. For each video length, the figure shows the median amount of prebuffering required as well

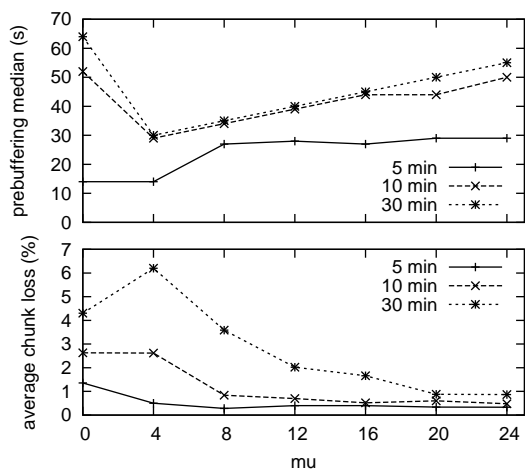


(a) The average chunk loss and the number of playing peers and seeders.



(b) The cumulative distribution of the prebuffering time.

**Figure 7. A sample run with half of the peers departing before playback is finished.**



**Figure 8. Average chunk loss and median prebuffering times against the size of the mid-priority set for several video lengths.**

as the average chunk loss for various values of  $\mu$ . Both the amount of chunk loss and the amount of required prebuffering increases with the video length for all the values tested. By changing the size of the mid priority set, the amount of chunk loss can be lowered but this results in a longer prebuffering time. Whether this is an acceptable trade-off depends on the video codec as well as the video content.

Like the previous experiments, the chunk loss in all runs was biased towards the beginning, when no seeders except the original one are available. The average chunk loss observed during that period is thus several times higher than the average over the full run. Compare for instance Figure 3(a), in which the movie length is 5 minutes and  $\mu = 4$ , against the respective datapoint in Figure 8. The average chunk loss over the whole run is 0.5%, although most of the loss occurs as an average chunk loss of around 3% for a period of 200 seconds.

## 6 Conclusions and Future Work

We have presented Give-to-Get, a P2P video-on-demand algorithm which discourages free-riding. In Give-to-Get, peers are encouraged to upload data to neighbours which prove to be the best forwarders. This policy hurts free-riders any time their neighbours have the opportunity to forward to well-behaving peers instead. When resources are abundant, such as when there are many seeders, free-riders can still join and watch the video at acceptable quality. This allows users with low upload capacity but enough download capacity such as ADSL users to join the system as long as it can support them.

Give-to-Get also has its limitations. First of all, it does not scale with the length of the video. Longer movies mean a higher chunk loss and so a lower perceived quality. Secondly, the algorithm relies on second-hand information—peers query others about the behaviour of their neighbours. If a user can easily generate multiple identities in the system, he can report about his own behaviour, which would make their free-riding go unnoticed. As future work, we plan to investigate whether these limitations can be removed, and whether the performance of Give-to-Get can be improved.

## References

- [1] E. Adar and B. Huberman. Freeriding on Gnutella. *First Monday*, 5(10), 2000.
- [2] N. Andrade, M. Mowbray, A. Lima, G. Wagner, and M. Rippeanu. Influences on Cooperation in BitTorrent Communities. In *Proc. of ACM SIGCOMM*, pages 111–115, 2005.
- [3] B. Cohen. BitTorrent. <http://www.bittorrent.com/>.
- [4] 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*, volume 3, pages 1467–1472, 2004.

- [5] Y. Guo, K. Suh, J. Kurose, and D. Towsley. P2Cast: P2P Patching Scheme for VoD Services. In *Proc. of the 12th World Wide Web Conference*, pages 301–309, 2003.
- [6] I. Keidar, R. Melamed, and A. Orda. EquiCast: Scalable Multicast with Selfish Users. In *Proc. of the 25th ACM Symposium on Principles of Distributed Computing*, pages 63–71, 2006.
- [7] H. Li, A. Clement, E. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. BAR Gossip. In *Proc. of the 7th USENIX Operating Systems Design and Implementation (OSDI)*, pages 191–206, 2006.
- [8] C. Liao, W. Sun, C. King, and H. Hsiao. OBN: Peering Finding Suppliers in P2P On-demand Streaming Systems. In *Proc. of the 12th Intl. Conf. on Parallel and Distributed Systems*, volume 1, pages 8–15, 2006.
- [9] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free Riding in BitTorrent is Cheap. In *Proc. of the 5th Workshop on Hot Topics in Networks*, 2006.
- [10] S. Sheu, K. Hua, and W. Tavanapong. Chaining: A Generalizing Batching Technique for Video-On-Demand Systems. In *Proc. of the Int. Conf. on Multimedia Computing and Systems*, pages 110–117, 1997.
- [11] A. Vlavianos, M. Iliofotou, and M. Faloutsos. BiToS: Enhancing BitTorrent for Supporting Streaming Applications. In *IEEE Global Internet Symposium*, 2006.