Deftpack: A Robust Piece-Picking Algorithm for Scalable Video Coding in P2P Systems

Riccardo Petrocco*, Michael Eberhard†, Johan Pouwelse*, Dick Epema*,
*Technische Universiteit Delft, Delft, The Netherlands, r.petrocco@gmail.com
†Klagenfurt University, Klagenfurt, Austria, michael.eberhard@itec.uni-klu.ac.at

***Abstract*—The volume of Internet video is growing, and is expected to exceed 57 percent of global consumer Internet traffic by 2014. Peer-to-Peer technology can help delivering this massive volume of traffic in a cost-efficient, scalable, and reliable manner. However, single bitrate streaming is not sufficient given today's device and network connection diversity. A possible solution to this problem is provided by layered coding techniques, such as Scalable Video Coding, which allow addressing this diversity by providing content in various qualities within a single bitstream.**

In this paper we propose a new self-adapting piece-picking algorithm for downloading layered video streams, called Deftpack. Our algorithm significantly reduces the number of stalls, minimises the frequency of quality changes during playback, and maximizes the effective usage of the available bandwidth. Deftpack is the first algorithm that is specifically crafted to take all these three quality dimensions into account simultaneously, thus increasing the overall quality of experience. Additionally, Deftpack can be integrated into Bittorrent-based P2P systems and so has the chance of large-scale deployment. Our results from realistic swarm simulations show that Deftpack significantly outperforms previously proposed algorithms for retrieving layered content when all three quality dimensions are taken into account.

I. Introduction

The consumption of all forms of Internet video is constantly growing, and is expected to exceed 57 percent of the overall Internet traffic by the end of 2014 [2]. Only the volume of video-on-demand (VoD) traffic already doubles every two and a half years, which is due to the higher bitrates used for High-Definition (HD) content and to the increasing number of consumers. As the rise in required bandwidth increases the distribution costs of video, efficient strategies for supporting video distribution are required. Thus, distribution mechanisms that aim to be scalable, fault-proof, and cost-efficient, like P2P systems, have become very popular [1, 21, 24, 25]. Additionally, videos are nowadays provided in different qualities to ensure that various types of end-user terminals can be supported. Layered coding schemes allow the encoding of content into several qualities within a single stream, providing the viewer with the best quality depending on his needs and capabilities. In this paper we propose a novel algorithm for downloading such layered content over P2P networks called Deftpack, which we have integrated into the Next-Share Bittorrent-based P2P system [12].

Due to the diverse capacities of end-user terminals for multimedia content like HDTV sets, notebooks, and mobile phones, and the heterogeneity of their network connections, the provisioning of content in a single quality is not sufficient anymore. Although the content can be provided multiple times in different qualities, such an approach is inefficient and leads to a waste of bandwidth. An alternative is provided by *layered coding schemes*, which provide the content in multiple qualities within a single bitstream. The source stream is encoded into several *layers* that can be retrieved independently and that can be used to increase or decrease the playback quality. This technology allows clients with devices of different types and with different connection technologies to access the same content, and, in a P2P system, to collaborate with each other in sharing the layers. With layered content, the problem of selecting the best sustainable quality is introduced, increasing the complexity of the *piece-picking* algorithm responsible for determining the best order to download the pieces of the required layers before their playback deadlines. As the download bandwidth available to a peer is limited, the piece-picking algorithm has to decide whether to focus on downloading pieces needed to ensure continuous playback, or on downloading pieces of higher layers to increase the playback quality. Deciding when to increase quality is not trivial as it involves the risk of not being able to maintain the new playback quality, thus reducing the overall quality of experience (QoE) [11].

Solutions for retrieving layered content over P2P networks have been extensively investigated in the last years [6, 14, 17, 19, 20, 23], but they do not simultaneously address the quality dimensions of reducing the number of stalls, minimising the frequency of quality changes, and maximizing the effective usage of bandwidth. In addition, our algorithm has been integrated into the open source P2P system Next-Share [12], which provides a BitTorrent-based solution for distributing layered content taking advantage of existing BitTorrent communities. Although the architecture and algorithms for supporting layered content within Next-Share are codec-agnostic, the system has been implemented using Scalable Video Coding (SVC).

The remainder of this paper is organized as follows. Section II provides an overview of the problems arising when selecting pieces of layered content for download. In Section III related work is discussed. In Section IV the design ideas and details of our new layered piece-picking algorithm are described. Section V describes the simulation settings used to evaluate our new piece-picking algorithm, while Section VI compares its performance to that of previous algorithms. Finally, Section VII concludes the paper.

978-0-7695-4589-9/11 $26.00 © 2011 IEEE
DOI 10.1109/ISM.2011.52

II. Problem Statement

In this paper we assume the video streams to be distributed by a P2P system to be encoded using the Scalable Video Coding (SVC) scheme. In SVC, the video bitstream consists of the H.264/AVC-compatible *base layer* and spatial or quality *enhancement layers*. The scalability of the video codec allows changes of the resolution and the visual quality by simply adding or discarding enhancement layers for the decoding process. While the base layer provides the minimum quality and can be decoded on its own, every enhancement layer requires all lower layers to be available for decoding. An example of a four-layer encoded stream is presented in Figure 1, where BL stands for base layer, and EL1-3 for the enhancement layers. The time slots on top of the figure represent the time intervals at the end of which the playback quality can be changed.

The main goal when designing a layered piece-picking algorithm is to provide the best possible quality for the following time interval and still ensure that all pieces are received in time for playback. Therefore, the P2P client needs to receive the pieces for every time slot from the lowest layer up to the desired playback quality to ensure that the content can be processed by the decoder. Thus, the order in which layers are received is crucial, as high-layer pieces cannot be processed if the corresponding pieces of the lower layers have not been received as well.

When layered content is transmitted through P2P systems, four major problems may arise. First of all, *stalling* may occur, that is, the decoder may not have data available for continuous decoding of the video stream. Stalling occurs when data from the base layer is not received in time for playback. This can happen if the peer's download capacity is not sufficient, or if the available download bandwidth is not smartly used to download the more important pieces first in highly dynamic environments such as P2P systems. This behaviour is particularly evident if the base layer is only prioritized over the enhancement layers of the same time slot, but not over the enhancement layers from previous time slots. An example of this situation is presented in Figure 1, where, given the playback position at time instant *t*, the playback will stall at the end of time slot *t* because the base layer in time slot *t+1* is not available (but the enhancement layers are).

Second, layered piece-picking algorithms may not make optimal use of the available bandwidth, introducing a *bandwidth usage* problem. These algorithms usually focus on the download of pieces within a sliding window that contains pieces for a limited number of time slots starting from the current playback position. However, if only the pieces for the near future are considered, the bandwidth may not be fully utilized if the network conditions are good and the peer has a download link capacity higher than what is needed for the playback of the highest available quality. As the network conditions in P2P systems are often very dynamic, the bandwidth should be utilized as much as possible, possibly for pieces further in the future.

Third, *bandwidth waste* may occur if pieces from higher layers are downloaded but are never used for playback. Piece-picking algorithms usually try to increase the quality as soon as possible, trying to fully utilize the available download bandwidth. Therefore it can happen that pieces are often selected for download even though their playback deadlines are already close. In such cases, the pieces may be received after their playback deadlines and thus be discarded because they are useless for the current playback. In Figure 1, if the playback would reach time slot *t+5* and the corresponding data of the enhancement layer EL1 for that slot would still be missing, (e.g., because it is requested from a slow peer), the player will only be able to display the base layer, wasting the bandwidth that has been invested in the download of the second enhancement layer (EL2).

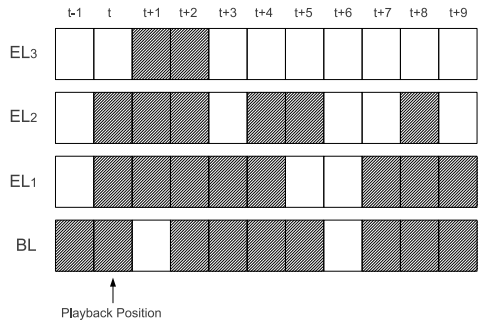


Fig. 1: An overview of the problems arising with SVC

The last problem regards the frequency of *quality changes* that occur during playback. Enhancement layer pieces are usually downloaded as soon as there is sufficient bandwidth, and as a consequence, the viewing quality may frequently be changed. Such frequent quality changes should be avoided, as they lead to a worse viewing experience than viewing the content at a lower, but constant, quality [11]. As an example, in Figure 1 it is better to avoid displaying the EL2 layer in time slot *t+8*, thus having a continuous playback of EL1 between *t+7* and *t+9*, rather than switching quality every time slot.

To address all of these problems, we have developed a new layered piece-picking algorithm that is described in Section IV.

III. Related Work

Over the last few years, P2P video on demand and live streaming have been widely investigated. The most recent work converges on the opinion that the usage of layered coding solutions such as SVC can be beneficial for P2P networks. Several solutions have been presented on how to design and implement layered coding support in P2P networks. In [19] a new approach that takes advantage of SVC as well as network coding is presented. Our approach differs since we only focus on the download algorithm relying on the existing Bittorrent [3] overlay and communities. In [16] an algorithm using a "zigzag scheduling" approach is presented. The zigzag-like piece-picking algorithm defines a zigzag priority order on the

pieces within a sliding window that contains the pieces with deadlines in the near future, prioritizing lower-layer pieces over higher-layer pieces. Using this zigzag order to sort the pieces according to their priority, the pieces are subsequently selected for download as long as there is sufficient download bandwidth available. Although the results show the advantage of using layered coding technology in P2P, their approach consists of a stream divided into 50 layers with a 8 kbit/s average bit-rate reaching a total of 400 kbit/s, causing many quality changes.

Recent studies show that frequently changing viewing quality has a bad effect on the user-perceived QoE [11]. To maximize the QoE, is better to stay at a lower quality rather than having frequent quality changes, even if it means loosing bandwidth efficiency. Therefore, one of the main goals of this paper is to show how to reach HD quality using as few layers as possible to minimize the number of quality changes during playback.

In [17] a complete new system has been designed and implemented focusing on robustness rather than on offering highly different bit-rates for different devices/connections. The paper focuses on the arriving requests of the peers from the network and presents a smart algorithm for scheduling. This design relies on the adoption of the algorithm by all the peers in the system and on the good nature of the seeders. Furthermore, there is no proposed algorithm for retrieving pieces in an efficient manner.

The deadline-based algorithm used in the Next-Share system [12], which is very simple and has a low complexity, downloads pieces according to their playback deadline, i.e., pieces with an earlier deadline are downloaded before pieces with a later deadline. This kind of algorithm is usually applied for single-layer streaming and ensures that the pieces are received in time for playback. When applied to layered content, it first downloads all of the base layer pieces according to their deadline, then continues downloading the pieces for the first enhancement layer, and so on, as long as there is bandwidth available for higher-layer pieces.

The KP-based piece-picking algorithm [9] is based on algorithms for solving the knapsack problem (KP) [18], which is a problem in combinatorial optimization. As the piece-picking of layered content and the knapsack problem are very similar, the KP-based algorithm reuses approaches to solve the KP for addressing the layered piece-picking problem. The algorithm first calculates the priority for all enhancement layer pieces based on a utility formula (see Equation 1), which takes the layer weights (lower enhancement layers are required for the decoding of higher enhancement layers), the remaining time until the playback deadline, and the download probability into account. After calculating the utility for all enhancement layer pieces within the sliding window that are not yet selected for download, these pieces are sorted according to their utility. From this sorted list the pieces are selected for download as long as there is download bandwidth available.

These last two algorithms provide good solutions for different scenarios, protocols and network conditions, and are considered for the design of our new piece-picking algorithm in the following section.

IV. Deftpack

In this section we describe our piece-picking algorithm, called Deftpack. Section IV-A describes the design of the algorithm while Section IV-B explains in detail the behaviour of its sliding window. Section IV-C describes how Deftpack addresses the problems stated in Section II.

A. The Design of Deftpack

The layered piece-picking algorithm presented in this paper, Deftpack, is based on a single-layer VoD algorithm to retrieve the base layer, and on the knapsack problem-based algorithm [18] for retrieving the enhancement layers. Deftpack divides the pieces to be downloaded into five sets, and decides from which one to select them, following the order of importance of the sets, which is indicated by the numbers in Figure 2. As the sizes of the sets are not relevant to their importance, Figure 2 does not show their real proportionality. Every set has a horizontal size, a range of pieces that represents a specific time window in the stream, and a vertical size, covering a number layers.

The horizontal boundaries of the high-priority set (set 1 in Figure 2), which is also called the sliding window, are determined by the current playback position in the stream and a certain size that might vary depending on the content duration. The right end of the high-priority set determines the beginning of the mid-priority set (set 2 in Figure 2), the size of which is set to four times the size of the sliding window. The remaining pieces, from the end of the mid-priority set until the end of the stream, define the low-priority set (set 3 in Figure 2). The *active layers* are defined as the layers that are currently used for playback, and they define the vertical boundary of the mid- and low-priority sets. In the example of Figure 2, the number of active layers is three.

Once all the pieces from the high-, mid-, and low-priority sets have been downloaded, Deftpack will select pieces from the lowest-priority set (set 4 in Figure 2). This gives a chance of improving the playback quality towards the end of the stream, increasing the QoE [11]. The past set (set 5 in Figure 2), represents all the pieces, for all the layers, from the beginning of the stream until the current playback position. Downloading pieces in the past set increases their availability in the swarm, and allows the user to watch the stream at its best quality once the download is finished.

Except for the high-priority set, pieces within all the sets are selected following the rarest-first approach. For the mid-, the low-, and the lowest-priority sets, this behaviour guarantees that the spare bandwidth, that is not being used for pieces from the sliding window, is invested in retrieving any pieces that will probably be needed in the future.

In the high-priority set, the pieces of the base layer are always downloaded in order, based on their playback deadlines, before pieces of the enhancement layers. Pieces of the enhancement layers are downloaded in decreasing order of

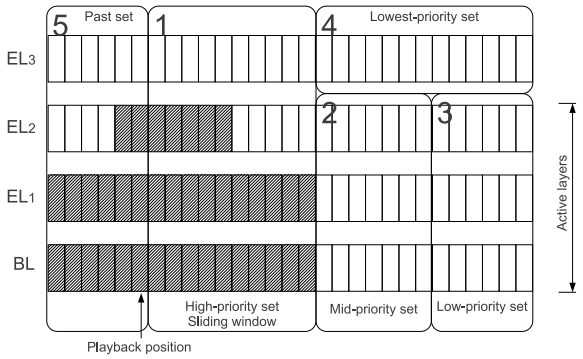


Fig. 2: The five priority sets distinguished by Deftpack. The grey pieces represent downloaded pieces.

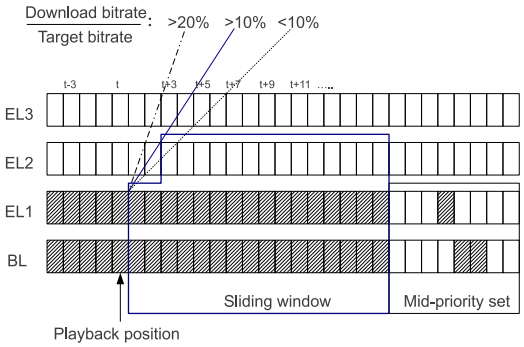


Fig. 3: An example of the dynamic window used by Deftpack when increasing quality.

their utility. The *utility* at time slot t_k of piece j (corresponding to time slot t_j) of layer i is defined as:

$$u_{ijk} = \frac{lw_i \times dp_i}{(t_j - t_k)^{\alpha}}, \tag{1}$$

where lw_i is the weight of layer i, dp_i is the probability that a piece can be downloaded under the current network conditions in time for playback [18], and α is a positive constant. In the simulations presented in Section VI, we investigate the simplest scenario where α is set to 1, and the weights of the BL, EL1, EL2, and EL3 are set to 4, 3, 2, and 1 respectively.

The low computational complexity of Deftpack when downloading only the base layer greatly reduces the computational requirements for hardware-limited devices, as these devices will never be able to display higher qualities due to their network connection or the resolution of their display. For devices that can also process the enhancement layers, the more complex KP-based algorithm is applied to display the best possible quality. The complexity of the algorithm is given by:

$$O(Deftpack) = \begin{cases} O(n) & \text{BL} \\ O(m \cdot n \cdot log(\max(m,n))), & \text{ELs} \end{cases} \tag{2}$$

where n and m are the horizontal and vertical size of the sliding window, respectively.

B. Dynamic sliding window

Previously proposed piece-picking algorithms for layered content use a sliding window to identify urgent pieces that need to be retrieved sooner than others for playback [16] [17]. This sliding window usually has a fixed horizontal size, related to a time range, and a fixed vertical size that includes all the available layers. Deftpack, on the other hand, uses a dynamic approach in order to optimize piece retrieval depending on the available resources.

To guarantee a near-zero-stall algorithm, and therefore continuous playback, the size of the sliding window is adjusted to ensure a stable download rate for the active layers. In contrast to other adaptive window approaches [15], where the window grows for fast peers and shrinks for slow peers to increase piece availability in the swarm, our algorithm will increase the window sizes for slow peers and shrink them for fast peers. As a consequence, slow peers avoid switching to a higher quality when their download speed is barely sufficient to guarantee the playback of the active layers, and fast peers sooner start downloading pieces of the enhancement layers.

We adjust the window size by monitoring the speed of data arrival of the pieces of the currently active layers as follows:

$$w_t = \left\lfloor \frac{k \times d_{al}}{b_{al}} \times w_{t-1} \right\rfloor, \quad b \leq w_t \leq w_{max}, \tag{3}$$

where w_t is the window size at time t, d_{al} is the average download speed of data of the active layers, b_{al} is the cumulative bitrate of the active layers, and k is a positive value that, starting from 1, increases by 0.1 for every 10 stalls. The window will never be smaller that a pre-defined size b (i.e., the player's buffer size in pieces), and never larger than the maximum preferred window size, w_{max}, to avoid retrieving all the remaining pieces in-order.

While the right end point of the sliding window is the same for all layers, the starting point is the same only for the active layers, as it might vary when Deftpack decides to increase quality. When increasing quality, it is unlikely to perform a smooth quality switch for the time slot immediately following the current playback position. Therefore, we move forward the window's starting point for the *target layer*, defined as the layer we are moving to, reducing bandwidth waste and increasing the probability of a successful quality switch. Deftpack will attempt to increase playback quality if the current download rate is higher than the *target bitrate*, defined as the playback bitrate that is achieved if the quality switch occurs. If the current download rate is 20% higher than the target bitrate, the window of the target layer will start one time slot after the current playback position, if it is between 10% and 20% higher, it starts two time slots later, and otherwise, it starts three time slots later. If the download speed suddenly increases, offering enough capacity to support an increase in quality of more than one layer, the same policy

is applied to all the desired layers.

Figure 3 presents an example of Deftpack increasing the quality from EL1 to EL2, the target layer. Here, the download rate is between 10% and 20% higher than the playback rate of BL+EL1+EL2, and therefore a gap of two time slots is set. As a second example, if in Figure 3 at instant *t*, the download rate would support the playback of all four layers, the window for EL2 will start at *t+3*, while the window for EL3 will start at *t+5*. This behaviour ensures a minimum amount of wasted bandwidth when increasing quality.

C. Meeting the Design Goals

This section describes how the problems described in Section II are addressed by Deftpack.

The occurrence of *stalling* events has been greatly reduced by prioritizing the base layer, and by adjusting the size of the sliding window. This behaviour ensures that a continuous playback is guaranteed even when the available download bandwidth is low. Additionally, it increases the availability of pieces from the base layer in the swarm, reducing the risk of stalling caused by the high competition that can occur with rare pieces. Deftpack optimizes the *bandwidth usage* of a peer by avoiding investing available bandwidth beyond the active layers, which might risk downloading pieces that might never be used for playback. As the algorithm downloads not only the pieces within the high-priority set, but also uses the remaining bandwidth to download pieces from the sets presented in Section IV-A, the download link's capacity is always fully used. This behaviour reduces the overall *bandwidth waste* of a peer. Furthermore, it is also reduced by the fact that our piece-picking algorithm only downloads pieces that can be realistically downloaded in time for playback. Frequent *quality changes* are avoided when possible as our piece-picking algorithm monitors the download bandwidth and only decides to switch to higher layers if the next enhancement layer can be sustainably downloaded. Utilizing this conservative approach, switches to higher qualities occur rarely and only when sufficient download bandwidth is available, while switches to lower layers only occur if the download bandwidth suddenly decreases.

Finally, it should be noted that Deftpack does not try to find the best possible solution considering each of the problems separately. Instead, the algorithm tries to find a feasible solution for each of the problems while ensuring that all of the other problems are still considered.

V. Experimental Setup

In this section we describe the experimental setup used to evaluate the performance of our Deftpack algorithm and for comparing it to three existing algorithms, ZigZag, KP-based, and NS-Core, described in Section III.

A. Simulator Setup

To perform an evaluation of our algorithm and compare it with the existing solutions, we have modified the discrete event-based Microsoft Research BitTorrent simulator [4, 7], implementing only the investigated piece-picking algorithms. For the results presented in this paper we assume that all peers are connected to each other and that for all the investigated algorithms the playback will start as soon as an initial set of pieces has been downloaded. Throughout the rest of the paper we will refer to this initial set of pieces as the buffer.

TABLE I: Peers bandwidth distribution

Class	**Distribution** [%]	**Downlink** [kbit/s]	**Uplink** [kbit/s]
DSL1	21.4	768	128
DSL2	23.3	1500	348
Cable1	18	3000	768
Cable2	37.7	10000	5000

TABLE II: Layer scheme used in simulations

Layer	**Layer bitrate** [kbit/s]	**Cumulative bitrate** [kbit/s]
BL	400	400
EL1	400	800
EL2	800	1600
EL3	1600	3200

For the peers participating in the swarm we show in Table I the distribution of their network capacities, dividing them into classes. The bandwidth distribution in Table I is based on the results of measurements [5, 8] and traces [13]. Peers slower than DSL1 connections have been omitted, adding their ratio to DSL1 peers, since recent measurements from public and private communities [10] have shown that the average Internet connection speed has increased significantly in the last few years. In Table II we show the bitrates of the layers used in the simulations.

B. Scenarios

In our simulations, we assume a peer leaves the swarm if either it has completely downloaded all the layers, if its playback has reached the end of the stream, or if its overall participation time in the swarm since its playback started exceeds the video duration by 50%. This occurs especially with slow peers when they experience too many stalls during the playback. If a peer experiences more than 50% of stalling time but its playback is currently progressing, it will leave the swarm at the next stall occurrence.

We focus on the behaviour of the investigated algorithms in scenarios with only one seeder, the initial content provider, forcing the peers to cooperate with each other. We also present results of running over-seeded scenarios, in which the initial seeders can support the swarm bandwidth demand. The scenarios we use to evaluate the performance of Deftpack can be grouped into three categories, depending on the peer arrival distribution and arrival rate.

In the *steady state* scenario, 500 peers join the system according to an exponential interarrival-time distribution during the first 30 minutes with an arrival rate of 0.11 peers per second. After that time, every peer that leaves is immediately replaced by a peer of the same class, thus maintaining a

TABLE III: Simulation settings

Parameter	Value
Video duration	60 minutes
Piece size	128 KB
Upload slots	5
Initial window size	20 pieces
Maximum window size	50 pieces
Buffer size	10 pieces
Seeders upload capacity	6 Mbit/s

constant number of peers. In this scenario, the content provider is the only initial seeder, causing the need for peers to collaborate with each other. We simulate 10 hours of operation, which gives results that are sufficient to draw conclusions.

In the *flashcrowd* scenario, the content provider is the only initial seeder, and peers arrive at an exponentially decaying rate, starting at a very high arrival rate. This scenario simulates a critical situation for a P2P system with a large number of peers joining at roughly the same time, e.g., to consume the broadcast of a live event. The arrival rate is given by:

$$\lambda(t) = \lambda_0 e^{-\gamma t}. \quad (4)$$

In our simulations we set $\lambda_0 = 10$ and $\gamma = 1/150$. With these parameter values, 1500 peers join the swarm during the first 50 minutes.

In the *over-seeded* scenarios, hundreds of seeders never leave the system, reducing the need for collaboration between peers. The first two of these are equal to the steady-state scenario but with 150 and 300 seeders instead of 1. The third of these is equal to the flashcrowd scenario but with 150 initial seeders.

The common properties of all the scenarios are presented in Table III. When simulating the same scenario with the four algorithms, we use the exact same arrival pattern of peers. Furthermore, the three algorithms against which we compare will download pieces in a rarest-first fashion from the entire stream if no piece can be selected from the sliding window.

C. Performance Metrics

In our performance evaluation, we use the following metrics. The *total playback bitrate* is defined as the cumulative playback bitrate of all the peers viewing the content at every second of the simulation run. The *total wasted bitrate* is defined as the bandwidth spent downloading data from the enhancement layers that is never displayed, which occurs when during playback a piece of an enhancement layer is needed but has only been partially downloaded. We only consider the wasted bandwidth for pieces from the sliding window, ignoring partial pieces that have been selected the from other sets.

If a peer does not download a base layer piece before its deadline, the playback will stall. For every stall occurrence, the playback pauses and resumes only after the initial buffer has been restored. We also monitor the *stalling time*, defined as the time spent by a peer waiting for the playback to resume from a stall. We report the occurrence of *stalls over time* to show how it influences the playback bitrate and pieces availability in the swarm. As stall occurrences are closely related to the availability of pieces of the base layer to be processed by the decoder, we monitor the *piece availability* of these pieces within the sliding window. Finally, we define the *abort percentage* as the percentage of peers aborting the download because they stalled for 50% of the video duration. We consider this as the most important metric to determine the QoE.

VI. Experimental Results

In this section we evaluate the performances of Deftpack by means of simulations. In Section VI-A we present the results for the steady state scenario, in Section VI-B for the flashcrowd scenario, and in Section VI-C for the over-seeded scenarios.

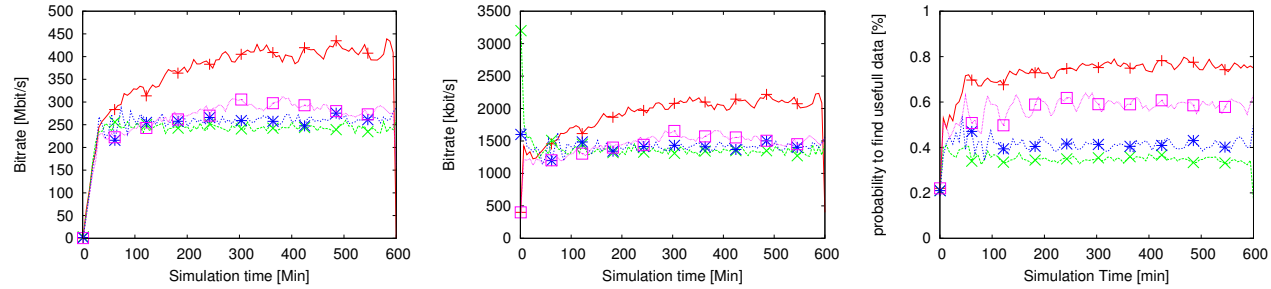
A. Steady State Scenario

Figure 4 presents the behaviour of the algorithms in a steady state. Figure 4(a) shows the cumulative playback bitrate for all the peers participating in the swarm for every second of the simulation run. Without the condition for moving to a higher quality, Deftpack would drastically outperform the other algorithms, but this would come at the cost of more frequent quality changes, and therefore, a more unstable playback. Figure 4(b) shows the average playback bitrate of all the peers in the swarm. It is interesting to noti how the ZigZag and the KP-based algorithms start at very high rates, 1.6 and 3.2 Mbit/s respectively, as all the layers are retrieved before the initial buffer is filled, and then quickly stabilize at lower rates. On the other hand, the NS-Core and Deftpack algorithms start at a lower rate, 400 Kbit/s, but reach a higher stable rate during the steady state. We do not show the wasted bandwidth, as it turns out to be very minimal (0.3% for the NS-Core and KP-based algorithms, 0.2% for Deftpack, and 1.1% for the ZigZag algorithm).

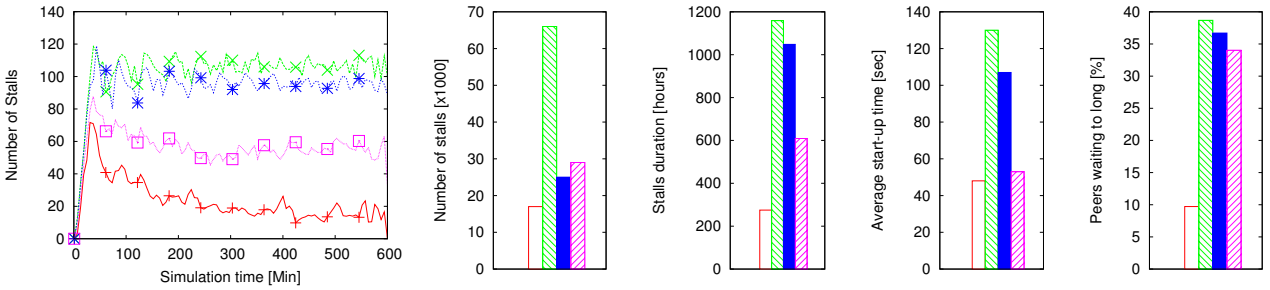
In Figure 4(b) shows that with Deftpack, the availability of pieces of the base layer in the high-priority set is much higher than with the other algorithms, leading to a much lower number of stalled peers over time (Figure 4(c)), a (much) lower total number of stalls (Figure 4(e)), and much less time spent by the peers refilling their buffers after stall occurrences (Figure 4(f)). Figure 4(g) shows the average start-up delay, while Figure 4(h) shows the percentage of peers leaving the swarm after waiting for more than 50% of the video duration.

It has to be noted that the KP-based algorithm provides the best playback quality to fast peers, but the poorest quality to slow peers. The main idea behind Deftpack is to be able to serve a broader audience, even if fast peers consume the content at a lower quality, giving a greater chance of a decent playback to slow peers.

Clearly, Deftpack outperforms the compared algorithms, optimizing the distribution of the base layer (Figure 4(c)), reducing the time for resuming the playback (Figure 4(f)), and therefore having the largest number of peers reaching the end



(a) The cumulative playback bitrate of all the peers (b) The average playback bitrate of all the peers (c) The availability of pieces of the base layer in the high-priority set



(d) The number of playback stalls (e) Number of stalls (f) Stall duration (g) Start-up delay (h) Abort percentage

Fig. 4: The performance of the four investigated algorithm in the steady state scenario. Deftpack (—□—) ; KP-Based (-⁎-▧-); ZigZag (-⁎-■-); NS-Core (··□··▨).

of the stream (Figure 4(h)), while still providing the highest QoE (Figures 4(a) and 4(b)).

B. Flashcrowd Scenario

Figure 5 shows how the investigated algorithms perform in the flashcrowd scenario; only four graphs are showed for the lack of space. Figure 5(a) shows the cumulative playback rate of all the peers in the swarm. During the first 50 minutes, after which no more peers join the swarm as then the arrival rate is effectively equal to zero, the KP-based and the ZigZag algorithms reach a high playback rate. This behaviour is caused by the way the buffer is filled before the playback can resume after a stall, as those two algorithms will download pieces from enhancement layers before the buffer has been successfully filled. As a consequence, during the same time period, they present 70% more stalls than Deftpack and the NS-Core algorithm.

While the cumulative playback rate decreases as peers are leaving the system (Figure 5(a)), the average playback bitrate increases as the initial seeder can support their bandwidth request (Figure 5(c)). Figure 5(b) shows how almost all the peers leave the swarm because of waiting too long when using the KP-base and the ZigZag algorithms. The NS-Core algorithm performs well compared to Deftpack, reaching a higher average playback rate (Figure 5(c)), but with a relatively higher bandwidth waste (40%), and for fewer peers (Figure 5(a)). In comparison to Deftpack, the NS-Core algorithm has a lower number of stall events (10%), but they last longer (Figure 5(d)), which causes roughly double the number of peers to leave the system before completing their playback (Figure 5(b)).

C. Over-seeded Scenario

In this section, the simulation results for the over-seeded scenario are presented to show how Deftpack performs in comparison to the other algorithms in non-critical situations. The results of those simulations are presented in Table IV. The most important results are the cumulative playback bitrate and the abort rate (columns *PlaybackBW* and *Abort* in Table IV).

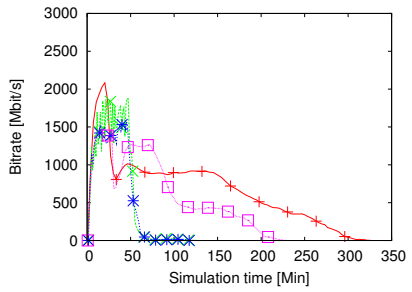
Deftpack clearly outperforms the other algorithms in steady state over-seeded scenarios. In the flashcrowd scenario with 150 seeders, Deftpack seems to perform generally worse than the other algorithms as it has a lower cumulative playback bitrate and longer start-up times. However, these results are misleading as with Deft pack, most of the peers actually reach the end of the stream (see the *Abort* column in Table IV), even if displaying a lower quality. This scenario can be compared with the one presented in Section VI-B where it is explained how this behaviour provides a higher QoE for the viewer.

VII. Conclusion

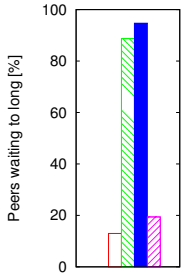
In this paper we present a new piece-picking algorithm for downloading layered content in P2P networks, called Deftpack, which reduces stalling of the video playback while ensuring that the user receives the best possible quality. In our simulations, Deftpack is compared to other piece-picking

Scenario	Algorithm	PlaybackBW [Mb/s]			WastedBW [%]			Stalls [x1000]			Stall time [x1000s]			Abort [%]			Start-up [s]		
Steady State 150 / 300 seeders	Deftpack	325	/	348	0.21	/	0.34	15	/	12	622	/	485	7	/	5	36	/	32
	KP-Based	251	/	257	0.39	/	0.35	129	/	95	4177	/	3940	44	/	60	105	/	142
	ZigZag	198	/	233	1.77	/	1.36	76	/	55	4439	/	3747	62	/	63	119	/	129
	NS-Core	280	/	303	0.31	/	0.29	21	/	17	1237	/	905	22	/	17	36	/	32
Flashcrowd 150 / 300 seeders	Deftpack	369	/	622	0.21	/	0.10	15	/	8	657	/	358	10	/	4	68	/	43
	KP-Based	572	/	763	0.53	/	0.30	75	/	44	3056	/	2301	93	/	62	211	/	188
	ZigZag	654	/	631	0.48	/	1.14	13	/	18	805	/	2148	19	/	61	54	/	167
	NS-Core	605	/	504	0.43	/	0.17	12	/	11	789	/	568	38	/	14	46	/	43

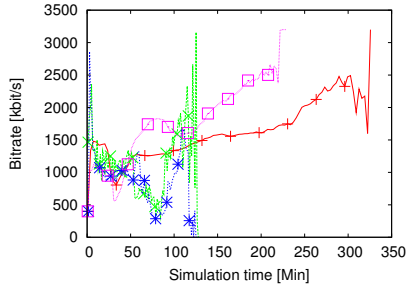
TABLE IV: Simulation results of the four investigated algorithms in the over-seeded scenarios. (Best results are highlighted.)



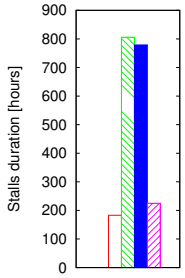
(a) The cumulative playback bitrate of all the peers



(b) Abort percentage



(c) The average playback quality from all peers



(d) Stall duration

Fig. 5: The performance of the four investigated algorithm in the flashcrowd scenario. Deftpack (—□—) ; KP-Based (-·×·-▧); ZigZag (-·✱·-■); NS-Core (··□··▨) .

algorithms in cases in which the content is under- and over-provisioned. In the under-provisioned scenarios, simulating real-world situations, Deftpack outperforms the other algorithms by reducing stalls and providing the highest QoE to most of the peers. For the over-provisioned scenarios, in which multiple seeders provide more than sufficient bandwidth to all peers, Deftpack clearly outperforms the other algorithms by providing the best average playback bandwidth while having the majority of the peers reaching the end of the playback.

Throughout all the simulations, it is shown how Deftpack allows peers with heterogeneous connections to share the same content in an efficient way. Overall, Deftpack significantly outperforms existing solutions, reducing the number of stalls, and providing the best overall QoE to the user. Furthermore, Deftpack has been integrated into the Next-Share system [12], which is to our knowledge the first open-source P2P system with full SVC support.

Acknowledgment

This work is supported in part by the European Commission in the context of the P2P-Next project (FP7-ICT-216217) [22].

References

[1] Akamai. http://www.akamai.com.
[2] Cisco. Cisco visual networking index: Forecast and methodology, 2009-2014. White paper, June 2009. http://www.cisco.com.
[3] B. Cohen. Bittorrent protocol 1.0. http://www.bittorrent.org.
[4] A. R. Bharambe et al. Analyzing and improving a BitTorrent networks performance mechanisms. In *Proc. of INFOCOM'06*, April 2006.
[5] C. Huang et al. Can internet video-on-demand be profitable? *SIGCOMM Comput. Commun. Rev.*, 37:133–144, August 2007.
[6] K. Lin et al. An optimized P2P based algorithm using SVC for media streaming. *ChinaCom2008-MCS*, May 2008.
[7] L. D'Acunto et al. Peer selection strategies for improved qos in heterogeneous bittorrent-like vod systems. ISM '10, pages 89–96, Washington, DC, USA, 2010.
[8] M. Dischinger et al. Characterizing residential broadband networks. In *Proc. of IMC'07*, IMC '07, pages 43–56, NY, USA, 2007. ACM.
[9] M. Eberhard et al. Knapsack problem-based piece-picking algorithms for layered content in peer-to-peer networks. In *AVSTP2P'10 WS Proc.*, AVSTP2P '10, pages 71–76, New York, NY, USA, 2010. ACM.
[10] M. Meulpolder et al. Public and private BitTorrent communities: a measurement study. In *Proc. of IPTPS'10*, IPTPS'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.
[11] M. Zink et al. Subjective impression of variations in layer encoded videos. In *Intl. WS on QoS*, pages 137–154, 2003.
[12] N. Capovilla et al. An architecture for distributing scalable content over peer-to-peer networks. In *MMEDIA'10*, pages 1–6, June 2010.
[13] S. S. Krishna et al. A measurement study of peer-to-peer file sharing systems. In *Proc. of MMCN'02*, 2002.
[14] T. C. Lee et al. Live video streaming using P2P and SVC. In *Proc. of MMNS08*, pages 104–113, 2008.
[15] Y. Borghol et al. Toward efficient on-demand streaming with BitTorrent. In *Proc. IFIP/TC6 Networking'10*, May 2010.
[16] Y. Ding et al. Peer-to-peer video-on-demand with scalable video coding. *Computer Communications*, 33(14):1589–1597, September 2010.
[17] Z. Liu et al. LayerP2P: Using layered video chunks in P2P live streaming. *IEEE Trans. on MM*, 11(7):1340–1352, August 2009.
[18] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990.
[19] S. Mirshokraie and M. Hefeeda. Live peer-to-peer streaming with scalable video coding and networking coding. In *Proc. of MMSys10*, pages 123–132, 2010.
[20] K. Mokhtarian and M. Hefeeda. Efficient allocation of seed servers in peer-to-peer streaming systems with scalable videos. In *IWQoS'09*, pages 1–9. Charleston, July 2010.
[21] PPSTream. http://www.ppstream.com.
[22] The P2P-Next Project. Fp7-ict-216217. http://www.p2p-next.org.
[23] R. Rejaie and A. Ortega. PALS: peer-to-peer adaptive layered streaming. In *NOSSDAV'03 Proc.*, NOSSDAV '03, pages 153–161, New York, NY, USA, 2003. ACM.
[24] SOPCast. http://www.sopcast.org.
[25] TVAnts. http://www.tvants.com.